1

OPTIMIZATION TECHNOLOGY, INC.

SCIENTIFIC AND TECHNICAL REPORTS SUMMARY FINAL REPORT CDRL A002

Environment For Test And Analysis Of Distributed Software (ETADS)

September 28, 1994

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army policy or position unless so designated by other official documentation.

Under Prime Contract
DASG60-94-C-0036

950 Explorer Boulevard
Suite 3C
Huntsville, Alabama 35806
205-922-1286

SECURITY CLASSIFICATION OF THIS PAGE OPTIMIZATION TECHNOLOGY, INC. CONTRACT DASG60-94-C-0036

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None ?
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Optimization Technology, Inc. / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) 950 Explorer Blvd, Suite 3C Huntsville, AL 35806	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION US Army Space & Strategic Defense Command / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DASG60-94-C-0036
8c. ADDRESS (City, State, and ZIP Code) P.O. Box 1500 Huntsville, AL 35807-3801	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Environment for Test and Analysis of Distributed Software (ETADS) Final Report

12. PERSONAL AUTHOR(S)
R.C. Cox, B. Allen

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
	FROM 3/28/94 TO 9/27/94	940927	120

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Distributed Software Testing, Software Testing Environment, Control Flow Analysis, Data Flow Analysis, Timing Analysis, Software Instrumentation, Non-Determinism, Deterministic Repla

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
This final report describes results of the Phase I SBIR research effort to develop new software testing techniques capable of satisfying the demands of distributed real-time software environments. Traditional software testing techniques are inadequate for distribute systems due to such complicating factors as non-deterministic execution, real-time timing constraints, task interaction event sequencing, race conditions, etc. During the course of the Phase I effort, OTI investigate a wide variety of candidate technique. For each technique, OTI evaluated the feasibility and defined a research and development approach for the technique and required supporting technologies. Detailed comparison of the attributes of each technique gives the folloing prioritization of techniques: Deterministic Execution Testing, Timing Analysis, Control Flow Testing, Dependency Analysis, Network Communication Testing, Data Flow Analysis, Problem Tracking Facility, Mutation Analysis, Static Concurrency Testing. OTI's research approach was based on developing new functionality on top of an existing software testing environment to construct the Environment for Test and Analysis of Distributed Software (ETADS).

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

Table Of Contents

1.0 Introduction

1.1 Purpose

Based on concepts initiated in the mid-1970's, distributed computing currently provides a feasible solution for many large-scale, real-time military and commercial applications. Advances in computer hardware, communications, interconnect/network technology (both short and long haul), and software engineering during the 1980's provided the resources to reduce these concepts to reality. Many distributed computing systems are operative today, and it is likely that as we move toward the 21st century distributed computing solutions for large-scale real-time applications will increase tremendously. This is evident from high bandwidth medical/hospital applications to the distributed computing system proposed for the Space Station. The Information Highway advocated by the current administration presents far-reaching challenges for large-scale, real-time, distributed computation both in the military and commercial sectors. However, in reaching the current state-of-the-practice in distributed computing, hardware engineering has always led, and continues to lead, software engineering.

Infrastructure tools to provide software engineering support for distributed systems development are nonexistent. Although there are numerous automated tools for insuring quality and reliability in sequential applications, there are none targeted to the unique and challenging problems associated with distributed systems. This is particularly true with regard to automated software testing. This is understandable in that acceptable solutions for providing the architectural functionality required of distributed software systems (e.g. control, data management, etc.) as well as conceptual models of techniques and methodologies for their design and implementation have just reached a state of early maturity over the past decade. These achievements were necessary to provide the foundation for formulation of the requirements for a distributed software test environment.

The importance and widespread use of large-scale, real-time, distributed software systems will continue to increase both in the DoD and commercial sectors. Automated testing technology supporting the delivery and maintenance of quality software is a key driver in the acceptance and full exploitation of this technology. The additional complexity of a test environment for distributed software systems must be recognized and an immediate initiative taken to develop such an environment. Without the specialized tools and techniques for testing large-scale, real-time, distributed software, the quality of such systems will suffer tremendously. As a result, there will be significant demand for such tools in both the public and private sectors as distributed computing continues to expand.

OTI is able to bring significant leverage to this difficult technical area. OTI is a recognized leader in software reliability and quality tools and methodologies, including software testing, metrics, reverse engineering, and maintenance. OTI developed, fielded, and applied a software quality analysis tool called SADCA (Static and Dynamic Code Analyzer). SADCA has been successfully applied to more than a dozen DoD software systems. It is an integrated software test and analysis environment designed to support the software engineering process from design through testing and maintenance. As proven technology it provides a platform to launch a distributed software test environment. SADCA is expected to significantly reduce the cost and lead time for providing a fully operational demonstration capability in recognition of the fact that: 1) the majority of the existing attributes in SADCA will be required for a new environment, thus avoiding duplication of effort;

and 2) new features will interface to and utilize data and control functionality from many existing SADCA features, thus reducing cost and lead time for the delivery of an operational product.

Additionally, OTI is in the second year of a three year research and development program for the USAF Rome Laboratories with the objective of developing testing techniques for Highly Parallel Software. As was indicated in our Phase I Proposal, appropriate technology from the Rome Laboratories program will be transferred to this program as a part of the Phase II effort. Finally, it should be emphasized that OTI relies on inputs from our software test engineers to derive useful and realistic attributes for our test tools. Furthermore, these personnel have been and will continue to be members of the research and development teams for our software test tools and related products.

The purpose of this final report (CDRL A002) is to describe the research results achieved in the development of software testing support for distributed software under the Phase I SBIR contract DASG60-94-C-0036 under the direction of USASSDC. The technology developed and documented under this program has been approached from the perspective of distributed, large scale, real-time, parallel systems. This final report describes a broad range of technical issues together with candidate solution approaches.

OTI's Phase I results, as presented in this final report, demonstrate the feasibility of an automated test environment for distributed software and, we believe, represent a justification for a Phase II research and development effort. At the conclusion of our proposed Phase II Effort, OTI will deliver a new release of the SADCA software test tool which will contain features and functionality that will extend its capability to static and dynamic testing of Large-Scale, Real-Time, Distributed Software Systems. As in the current version of SADCA, these features will not only support development testing and maintenance but regression testing, reverse engineering and reuse. This version of the tool will be referred to in this report as ETADS, Environment for Test and Analysis of Distributed Software. OTI has received a Corporate funding commitment from the Board of Directors to commercialize and incorporate all successful technology from the Phase II effort as a Phase III effort.

1.2 Scope

The primary focus of this document is concerned with the research results achieved during the SBIR Phase I effort. However, additional background information is provided in order to improve the degree to which this final report serves as a stand-alone document. For example, since the Phase I effort and subsequent proposed Phase II effort is predicated on the past successes of the SADCA technology developed by OTI, the section titled "SADCA Technology Background" is provided to illustrate the structure of the SADCA tool and to describe the static analysis and dynamic analysis features provided for the sequential software domain. In addition, another section titled "Distributed Software Background" is presented to provided for a common frame of reference in describing the Phase I results.

1.3 Document Overview

Chapter 2 provides an executive overview of the Phase I results. A summary description of the SADCA technology upon which this Phase I research is predicated is presented in Chapter 3. Chapter 4 presents a background discussion of the Distributed Software domain and provides a common frame of reference for further discussion of the Phase I research results.

Chapter 5 presents detailed descriptions of the Phase I research results. The results are organized into subsections, one for each ETADS technology area. Each subsection is further divided to separately present the objectives and benefits, the technical issues, and the details of the research results. Chapter 6 provides a summary discussion of the results and conclusions. As a major part of the conclusions, this section presents a prioritization of the technical results in a succinct tabular form.

2.0 Executive Overview

Research and development under the Phase I effort has led OTI to the specification of the Environment for Test and Analysis of Distributed Software (ETADS) architecture described below. This section of the final report is designed to summarize the characteristics of the ETADS target architecture without expounding upon all of the many details of implementation. Further elaboration of the details can be found in the subsequent sections of this final report. Detailed descriptions of individual techniques supporting a particular testing technology can be found in Section 5.

2.1 Objectives/Benefits

The ETADS objective is to facilitate the process of testing distributed real-time software. The problem domain involves geographically distributed computing resources interconnected via LAN/WAN technology and may involve real-time data processing requirements. Furthermore, the software testing technology advanced under this research effort explicitly maintains the objective of scalability to large-scale systems.

The following table lists the principal functional areas advanced under the Phase I research area and is organized along major functional capabilities to be included in the ETADS tool. However, there is also a collection of underlying technology concerns that must be addressed to support these functional areas. These related supporting technology concerns are also listed below. The ETADS objectives are to provide additional automated tools support for each of these technology areas.

- Major Technical Areas
  - Static concurrency and data flow analysis
  - Timing analysis
  - Deterministic execution testing
  - Dependency analysis
  - Mutation analysis
  - Control flow testing
  - Network communications testing
  - Problem tracking and reporting
- Related Supporting Technology Concerns
  - Distributed clock synchronization/compensation
  - Time-stamp generation
  - Minimization of probe effects
  - Trace data compression
  - Trace file management

Static concurrency and data flow analysis facilitates the identification of structural problems with the software under test (SUT). The focus of this activity is to identify, and subsequently remove, errors and potential errors without the need to execute the software. For example, concurrent software may be inadvertently designed such that one or more processes can enters an infinite wait state because it is waiting for an event that will never occur. Alternatively, concurrent software in which shared memory is supported may be configured such that data race conditions exist where one process may overwrite the results generated by another process before those results are ever used. The benefit to software quality through the identification and removal of such errors is clear and automated analysis support in this area frees software development resources to be applied in other software testing activities. In addition, the results of the analysis can be used to construct a variety of informative reports that facilitate the human comprehension of the software's structure and potential behavior.

ETADS support for timing analysis will provide for visualization of interprocess timing relationships to facilitate the verification and understanding of complex system behavior. Various alternative views of the timing relationships from different perspectives are required to facilitate the identification of anomalous timing relationships and improve human comprehension of observed system behavior. For example, incorrect or unexpected system behavior may be identified through the graphical display of event orderings which reveals the actual sequencing of events as opposed to the expected sequence of events. In addition, observation of system performance can be graphically displayed with respect to the system timeline in comparison to the real-time constraints. Such graphical comparison can provide a succinct report in which identification and recognition of an error condition (in the form of real-time constraint violation) is immediate.

Development of support for deterministic execution testing addresses one of the primary problems encountered in testing distributed real-time software: non-deterministic behavior of the SUT as a consequence of variations in timing. With non-real-time sequential software, the developer had the luxury of knowing that the software will always behave the same for each execution of given test case to produce the same output for the same input. In contrast, distributed real-time software may generate different results every time a given test case is executed since the precise sequence of instructions executed (i.e. the execution path) may be different for each test case if the times at which messages are sent and/or received is in any way different. The effect of this behavior is of particular concern to the software developer when attempting to identify the cause of an observed error since it may be difficult or effectively impossible to reproduce the precise set of input data and sequence of events leading to the error condition. Such non-deterministic behavior presents one of the most prevalent problems encountered in testing distributed real-time systems: difficulty in reproducing an error in response to a particular test case. Without special tools support to address the non-deterministic behavior within the system, the complexity of identifying the source of an error is much more severe.

Dependency analysis services are an important part of the basic ETADS package. The objective of this form of analysis is to focus on identifying and reporting the process dependency relationship (i.e. through process creation) between a parent process and its collection of child processes. During the analysis, it may also be possible to statically identify and report relationships between sibling processes (i.e. a collection of child processes having the same parent process). Further analysis and reporting based on trace data collected from software tests may reveal further dependency relationships that are only identifiable only at run-time (i.e. dynamically). Identification and reporting of process dependencies can be of great value in understanding the run-time behavior of the system as it provides a higher level abstract view of system operations in terms of the hierarchical relationships between processes.

Mutation analysis is an area of testing that has principally been applied to the sequential domain but which can be extended to benefit the distributed domain. In summary, the mutation analysis technique involves the insertion of specific changes/mutation(s) into the SUT in order to create a new version mutant of the SUT (i.e. a mutant) against which the test suite will be executed again to determine and evaluate the ability of the test suite to detect the presence of the mutation(s). This procedure is designed to enhance software reliability by supporting the evaluation of the thoroughness of the suite of software test cases followed by the improvement of the test case suite to the desired level of completeness (e.g. improvement by developing new and/or different test cases). The ETADS technology development focused on the identification of a sound strategy for integrating automated support for mutation testing within a common tool framework. Automated means can be devised, and have been defined, to facilitate the construction and execution of mutant versions of the SUT.

The objective with respect to control flow testing is to extend the test case coverage analysis capabilities currently provided in the sequential domain to provide similar capabilities for the distributed environment. In OTI's SADCA tool, coverage analysis capabilities provide the ability to identify which portions of the SUT has been exercised by a particular test case. A particular statement from the SUT is said to be "covered" if a test case causes the statement to be executed. In addition, SADCA measures coverage in terms of which True/False branches of decision statements have been executed, and within decision statements such as an "if" statement, SADCA measures which combinations of condition outcomes are exercised. Furthermore, SADCA technology supports the evaluation of coverage achieved over the complete sequence of test cases contained in the developer's test suite. The ETADS objective is to demonstrate the application of the multilingual test engine to the distributed environment and extend its coverage analysis capabilities.

The ETADS objective with respect to network communications testing is to further develop tools and techniques that focus on analyzing, reporting and visualizing system performance with respect to the network communications traffic. From this perspective, mechanisms can be defined that highlight the message patterns, communications bandwidth utilization, interprocess message patterns, interprocessor message loading, etc.

Another objective in the ETADS development is the integration of facilities for problem tracking and reporting, which have been defined and documented. This objective addresses the need to evaluate the effectiveness of new software analysis and testing services as they are being applied in the user's software development environment. By integrating the tracking and reporting service into the same tool environment, the user's of the ETADS tools can develop usage statistics as the tool's services are employed and can indicate which techniques are being employed with the greatest benefit to software quality.

Other secondary objectives have been identified and pursued which are not directly attributable to functions that are visible at the user interface level. However, these secondary objectives should not be construed as inconsequential. For example, the development of techniques to address distributed clock synchronization/compensation has also been pursued as a supporting technology objective of the Phase I research. Analysis of distributed system performance, whether automated or manual, generally requires the correlation of events with respect to time (e.g. in support of timing analysis) in order to establish the system behavior with respect to a global system timeline. Since the accuracy of the analysis performed is clearly limited by the accuracy of the underlying timing data, significant engineering resources must be applied to insure that the accuracy of the collected timing data is maximized within the limits imposed by the execution environment. Another supporting objective was the identification and documentation of trades and issues involved in time-stamp generation strategies employed in a fielded ETADS tool. Similarly, one must consider, as important secondary objectives, the definition and documentation of strategies for minimizing probe effects (i.e. the run-time effects of inserting the special data collection statements into the SUT) and the definition of a general strategy for accomplishing compression of trace data. These must be treated as important objectives since the implementation strategy can have significant impact on the real-time behavior of the SUT and such impact must be minimized.

The ETADS research focuses on developing software testing techniques suitable for each of these objectives and the development of automated tool support to mitigate problems associated with software testing within this domain.

2.2 Technical Issues

The fundamental technical issues addressed under Phase I research are driven by the following characteristics of the target environment: real-time execution, parallel process execution, heterogeneous computing architecture, and physical distribution of computing resources. Solution approaches for each of these technical issues were addressed in the Phase I SBIR research effort in support of the stated objectives and their associated benefits. It should be anticipated that the detailed solution approach for a given technical issue can support multiple objectives since a given objective represents a relatively high level of abstraction for which the implementation is typically dependent on the integration of wide variety of SADCA/ETADS techniques.

From the perspective of real-time software execution, the design of test tool operations must be greatly concerned with impacting the behavior of the experiment (i.e. the observed performance of the SUT). This aspect is a major issue as the ability to observe the experiment behavior must be supported by some form of probe. As will be further discussed in Chapter 5, ETADS technology is based on the application of software probes as opposed to hardware probes. However, due caution must be exercised in the insertion of software probes in real-time software as each probe introduces some degree of perturbation of the timing relations of the SUT. Tools and techniques are needed that provide maximum flexibility to the user across a broad range of computing environment. The user should be able to pick and choose when and where software probes are applied such that the probe effects are minimized for any given experiment.

The real-time software domain also presents software developers with severe difficulties with regard to the reproducibility of observed behavior for a specific test case. Because timing relationships are important in determining the precise behavior of real-time software (i.e. different paths may be executed as a consequence of minor variations in timing), the test personnel may execute the same test case multiple times and produce different results each time. As a consequence, identification and elimination of errors can be quite difficult. Tools and techniques, such as those advanced in the ETADS Phase I research, are needed to mitigate or eliminate the effects of non-deterministic behavior in real-time software.

The concurrent execution of multiple processes, communicating via message passing and/or shared memory, typically results in complex run-time behavior for which even top notch software engineers may have difficulty predicting and/or understanding all of the possible timing relationships between the various processes. Because there is generally no definitive structural relationship between code in different processes, observation of the parallel program behavior requires the observation of the timing relationships. Identification of such timing relationships requires the observation and recording of time-stamped events, where an event corresponds to the execution of some specific statement in the SUT. Recording time-stamped event data (otherwise known as trace data) throughout the execution of the test case provides the basis for post-execution analysis of the execution. Trace data can be input to experiment visualization tools to illustrate the observed behavior of the software. However, it is very important to recognize that the data collection and recording process can be very resource intensive with respect to both CPU time and secondary storage. Thus, there is major incentive to optimize the process to minimize the requirement for recording events in general and even more incentive to minimize the requirement for time-stamped events.

Targeting distributed computing environments encompasses additional issues beyond those of the stand-alone sequential domain. Of major concern to the design of ETADS tools support is the degree of increased clock uncertainties. The distribute computing resources may have clocks that are not well synchronized and performance may vary widely with respect to accuracy and drift. Furthermore, some computing resources may have no clock available. In addition, the topology of the distributed computing environment may have a major impact on system performance and may be dynamically reconfigured, thus further complicating the post-experiment analysis of time-stamped trace data.

The existence of heterogeneous computing architectures further complicates the distributed software testing by introducing variations in the availability and performance of various services and resources. Some of the variations may be observed with respect to memory capacities, I/Of, clock, and secondary storage. Furthermore, the construction of a distributed heterogeneous system may easily involve processes programmed in a variety of languages such as Ada, C, and Fortran. In addition, the communications subsystems incorporated in the system may involve any number of different communications media requiring multiple communications interfaces. Tools and techniques, such as those included in ETADS, are needed to mitigate the inherent complexity introduced through the integration of different computing resources.

2.3 Technical Approach

In general, the technical approach applied under the Phase I ETADS development can be described in terms of the following set of analysis steps applied to each of the technology area as described in the Phase I proposal. For each technology area identified for research (i.e. those candidate technologies described in the Phase I proposal):

- Define data requirements for desired analysis features and reports
- Define data collection requirements
- Define and analyze steps required to implement data collection
- Define alternatives for implementing data collection steps and assign priorities based on estimated effort for implementation
- Enumerate existing SADCA data collection capabilities
- Comparison of ETADS data collection requirements vs. SADCA capabilities
- Collation of documented software requirements and design elements

2.4 ETADS Architecture

The ETADS architecture is illustrated below in Figure 2.4-1. Comparison of this architecture diagram with that of existing SADCA technology gives an indication of the degree to which software reuse facilitates the definition and construction of such an ambitious tool. The ETADS tool will provide a capability for cross-platform static and dynamic analysis that does not exist in any other tool

The ETADS multilingual capability is a natural extension of the SADCA technology. The user's distributed program may be composed of a distributed collection of programs coded in different languages, each of which executes on its own host, communicating with other processes over the LAN/WAN as necessary.

The ETADS architecture has been refined to the point where it can provide automated support in each of the major functional areas described earlier as objectives. In most cases, ETADS has been successfully refined to the point where strong support can be implemented. However, in the case of static concurrency and data flow analysis, additional research is deemed necessary to identify additional alternative techniques to improve the reporting capabilities and reduce the growth in algorithm size with respect to the size of the system to be analyzed.

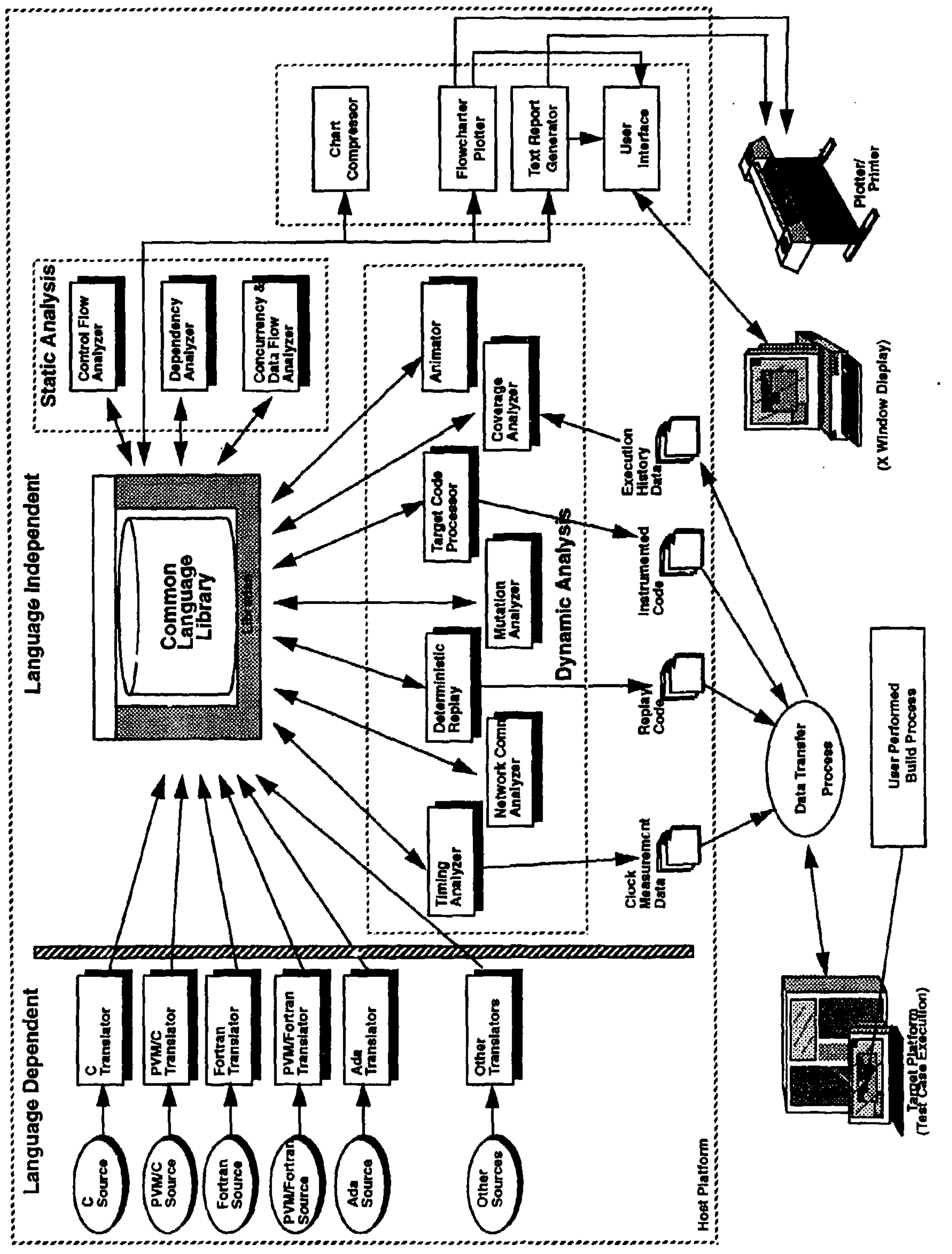


Figure 2.4-1 ETADS Architecture

3.0 SADCA/METAsoft Technology Background

The research effort undertaken during Phase I has been approached from the perspective of identifying feasible approaches to address the unique problems of the distributed real-time domain while constructing the required new technology on top of existing SADCA (Static and Dynamic Code Analyzer) technology. SADCA provides the means to demonstrate the various technology solutions outlined and facilitates the construction of an operational prototype of the ETADS tool in a cost-effective manner.

At each step in the research effort, OTI has been able to take maximum advantage of the experience gained from SADCA and subsequent commercial METAsoft (Multilingual Environment for Test and Analysis of Software) development to identify and document feasible strategies for addressing the new issues in the new problem domain. All of the SADCA technology has been successfully transitioned into the commercial METAsoft product. Demonstrations examples of SADCA/ METAsoft capabilities described herein are derived from METAsoft.

3.1 SADCA Purpose

SADCA is a multilingual automated software development, testing and maintenance tool. **SADCA** automates a large number of software life cycle activities, including (1) early detection of structural errors, (2) enforcement of coding standards, (3) systemization of a testing methodology, (4) quantification of test case coverage, (5) formalization of criteria to determine when to stop testing, (6) automated support for test case design, (7) minimization of regression testing activities and (8) visualization of code structure and data flow for use as automated generation method for code documentation. The structure of the SADCA tool is illustrated below in Figure 3.1-1.

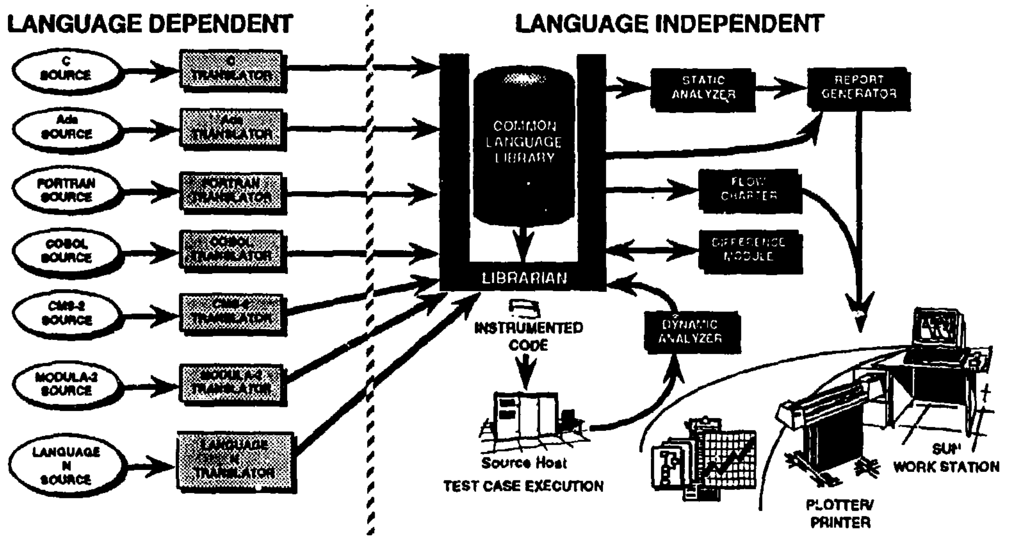


Figure 3.1-1 SADCA Architecture Overview

One of the major objectives of **SADCA** is the support of a variety of roles during the software development cycle. The tool provides statistics and reports that can be used by both the development organization (e.g., software developers, project managers, quality assurance personnel, etc.) and the acquisition organization (e.g., government managers, independent verification and validation personnel, etc.). **SADCA** has the ability to generate reports tailored to support varied levels of detail depending on the target audience. Its versatile Sun SPARC, X-Windows based interface allows for maximum tailoring to the user's specific needs. The common language interface to **SADCA** contains translators for the C, Ada and FORTRAN languages and has been designed with a language independent tool set facilitating a rapid and inexpensive path for the creation and linking of additional language translators.

SADCA possesses the following capabilities:

- **SADCA** contains a menu-driven, X-Windows interface, based on the OPEN LOOK™ standard, which allows the user to process source code written in Ada, C and FORTRAN into a common language format which can be operated upon by a single set of software testing and analysis tools. In comparison, METAsoft is being released with an upgrade to the Motif style.

- Through the unique ability of **SADCA** to process multiple languages into a single format for operation by a common tool set, the user may process multiple language sources all within a single program (e.g. **SADCA** can easily process Ada code which calls FORTRAN math libraries and graphics libraries written in C.)

- The **SADCA** data base allows the user to segment all source code processing and analysis into an easily maintained hierarchical structure composed of projects, the programs that fall under those projects and the files which make up each program.

- The **SADCA** Static Analyzer, in conjunction with data collected from the language translators, calculates a series of helpful metric reports, including McCabe's Cyclomatic Complexity Metric, Variable Cross-Reference, Variable Set/Use Anomaly, Halstead's Length, Vocabulary, Volume, Purity and Bug Metrics, Lines of Code/Comment Line Counts/Percentages, and Depth of Nesting Reports. Reverse engineering is supported by the generation of Data Flow reports, Charts Called reports, and Calling Charts reports.

- The **SADCA** Dynamic Analyzer, in conjunction with output generated from source code execution with test cases, calculates test case coverage data including Call Tree Coverage, Statement Coverage (C1), Decision Coverage (Branch/C2) and Multiple Condition Coverage.

- The **SADCA** Difference Facility allows the user to compare updated code versions with baselines and visually reference insertions, deletions and changes to the source along with all paths which may have been affected by these changes.

- The **SADCA** Results Facility allows the user access to static and dynamic analysis results in both a textual form suitable for inclusion in written reports and a graphical form for easy on-screen use.

- The **SADCA** Flow Charter provides the user with a graphical view of the source code processed. The user may: view static structure or coverage information at either the call tree level or the detailed flow graph level in the form of color coded graphs; view the static interface structure at all levels of abstraction from file and package to individual routines; compress covered graph segments to a single object to facilitate the location of testing yet to be accomplished; directly reference the originating source code on-screen in conjunction with the flow graph; view embedded function lists and traverse all code in the call structure from either the call tree table of contents or the expansion of function or procedure names.

- The SADCA Batch Processing Capability allows the use of all facilities normally accessed via the SADCA interactive user interface through an easily defined and identified batch script file.

3.2 Structure

The ETADS concept is defined as an extension of the existing SADCA technology and builds on the basic SADCA architecture. The following text describes each major SADCA CSC at a high level and the relationships between CSCs. Further description of the Static Analysis services and Dynamic Analysis services provided by SADCA will be presented in the following subsections.

The SADCA Architecture Diagram, as illustrated in Figure 3.2-1, graphically depicts the major internal data relationships between all SADCA CSCs and the major external relationships. This diagram contains a barrier that divides CSCs into two categories: language dependent and language independent. The CSCs on the left of the barrier are language dependent CSCs whose purpose is to translate specific target language source code into a common format to be stored in the SADCA Common Language Library. The CSCs to the right of the barrier are language independent and operate on data from the SADCA Common Language Library. Consequently, the majority of the SADCA functional components are language independent. Therefore, based on this design, the amount of additional SADCA code required to support new target languages is minimized. Each new target language only requires an appropriate language translator in order to translate target language source code into the SADCA Common Language Library. Static analysis is defined as the analysis that can be performed on the source code by itself without execution in order to determine a variety of static structural properties of the software under test (SUT). Dynamic analysis is defined as the analysis that can be performed as a result of target code execution.

The User Interface CSC (UI, CSC 1.1) provides a bidirectional communication path between the SADCA user and other CSCs. Appropriate X Window calls embedded in the UI CSC generate a meaningful graphic interface that allows the user to exercise the functionality of SADCA through various popup and pulldown menus. The user interacts with the graphic interface by manipulating the mouse pointer, mouse buttons and the keyboard. As a result of selected user requests, output from various SADCA CSCs is displayed in the interface. The UI CSC is invoked upon start-up of SADCA and exchanges data with the Text Report Generator, and Flowcharter as well as with the X Window display . Since control flow is not shown, it should be mentioned that the UI CSC exercises control over all other CSCs. By utilizing the power of advanced graphics, the SADCA graphic interface can be exploited in order to intuitively convey a large amount of analysis information in a relatively small physical area.

The Language Translator CSC (LT, CSC 1.2) contains the functionality to analyze the user's designated source code and derive a symbolic representation of the original code in a format that is suitable for display in a graphical form. Several of the steps involved in this CSC are common to compilers including lexical analysis and parsing. However, the language translator converts the source text into a symbolic representation instead of generating object code. In addition, this CSC is responsible for identifying and recording all of the points in the original source code at which software probes or instruments may be inserted to support the data collection process for the dynamic analysis of the user's program. For each identified instrumentation point, the Language Translator records in the database the language specific edit operations required to perform the probe insertion when constructing the instrumented version of the user's source code. During the performance of the translation activities, this CSC interacts with the Librarian CSC as necessary to

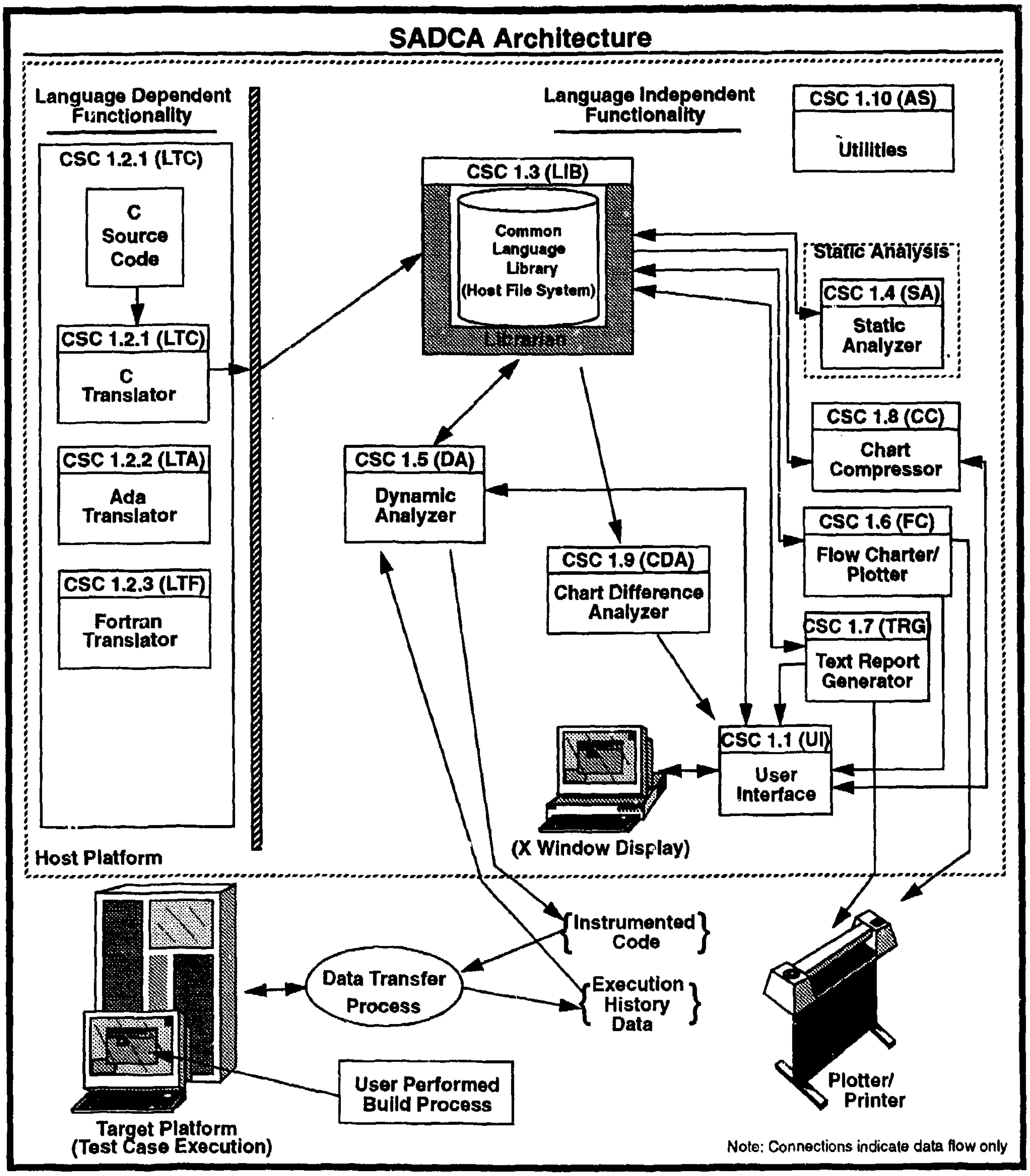


Figure 3.2-1 SADCA Architecture Diagram

store and retrieve information to and from the SADCA database. The LT CSC is further subdivided into lower level CSC's, one for each target language including: 1) LTC CSC 1.2.1 Language Translator C, 2) LTA CSC 1.2.2 Language Translator Ada, and 3) LTF CSC 1.2.3 Language Translator Fortran.

The Librarian CSC (LIB, CSC 1.3) serves as the interface to the SADCA Common Language Library. The SADCA Common Language Library provides a central location for all data shared between SADCA CSCs. Data contained within the SADCA Common Language Library consists of language independent data structures translated from language specific source code and intermediate data shared between other CSCs. Data contained in the SADCA Common Language Library can only be accessed through the Librarian interface. The Librarian interface is used by nearly all major CSCs.

The Static Analyzer CSC (SA, CSC 1.4) performs static analysis of translated target source code. This analysis includes the computation of various metrics. The SA CSC is invoked from the User Interface CSC. Results generated by this CSC are stored in the SADCA Common Language Library by the Librarian CSC.

The functionality of the Dynamic Analyzer CSC (DA, CSC 1.5) can be divided into two major categories: instrumentation of source code and processing of execution trace data. Based on input from the user, the DA CSC determines the necessary level of instruments to be inserted in the target source code. Once instrumented, the source code is moved to the target machine and executed. During execution, various instruments strategically placed throughout the code are also executed and significant pieces of data are recorded in a trace data file. Following execution the trace data file is moved to the SADCA host machine for processing. Trace data processing consists of extracting segments of data from the trace data file and encapsulating it in an appropriate data structure within the SADCA Common Language Library via the Librarian CSC. The DA CSC is invoked from the User Interface CSC. Records produced by this CSC are stored in the SADCA Common Language Library by the Librarian CSC.

The Flow Charter CSC (FC, CSC 1.6) provides the functionality required to construct flowcharts representing target code structure, execution path coverage, statement coverage, and chart differences (NOTE: the term "chart" is used as a general term referring to the composite entity, and its attributes, from which a flow chart diagram is drawn to represent the control flow structure of the corresponding subroutine from the originating code). The FC CSC is invoked from the User Interface CSC. Depending on input from the user, the FC CSC directs color flowcharts to a printer/plotter or to the User Interface CSC for display on the SADCA host machine's X Window terminal. Data required to construct flowcharts is obtained from the SADCA Common Language Library through the Librarian CSC.

The Text Report Generator CSC (TRG, CSC 1.7) provides the necessary functionality to generate reports from data housed in the SADCA Common Language Library. An example of one type of report would be coverage reports. Coverage reports contain formatted coverage data generated by the Dynamic Analyzer CSC in an easily readable report. This CSC is designed to allow the user to select the appropriate report based on the type of data reports required. The user has the ability to view the reports on the graphics display or generate hard copies. The TRG CSC is invoked from the User Interface CSC. Data required for reports is obtained from the Common Language Library through the Librarian CSC.

OTI OPTIMIZATION TECHNOLOGY, INC.
950 Explorer Boulevard • Suite 5C
Huntsville, Alabama 35806
TEL (205) 922-1288 • FAX (205) 922-1555

September, 28 1994

Defense Technical Information Center
Cameron Station
Alexandria, VA 22304-6145

Reference: Contract No. DASG60-94-C-0036, CLIN 0002, Data Item No. A002

Subject: Environment for Test and Analysis of Distributed Software (ETADS) Final Report

Dear Sir/Madam,

Enclosed please find the subject final report for your review. If you have any questions or comments please contact Robert C. Cox or the undersigned at (205) 922-1288 .

Sincerely,

Patricia G. Thompson

Patricia G. Thompson
Director, Contracts Administration

PGT/ps

An Affirmative Action/Equal Opportunity Employer

The Chart Compressor CSC (CC, CSC 1.8) provides the functionality necessary to reduce the size of flow charts displayed on the X Window's terminal. Chart compression allows several related, contiguous elements in a chart or graph to be represented by a single element, thereby allowing other relevant information to be displayed. Chart compression is used to conserve graphic display real estate and is used as an alternative, concise display mechanism for presenting test case coverage information. The CC CSC is invoked from the User Interface CSC. The CC CSC receives its input data from flowchart information stored in memory.

The Chart Difference Analyzer CSC (CDA, CSC 1.9) provides the automated service to compare two versions of a specific chart and identify the differences between them. In performing this chart comparison, the CDA CSC is performing a comparison of two subroutines and reporting the differences between them using a graphical presentation.

The Utilities CSC (AS, CSC 1.10) provides a broad collection of general utilities that support the operations of the other CSC's.

3.3 Overview of Services

Since SADCA technology has been successfully transitioned into the commercial world in the form of METAsoft, the following discussion of services describes the comprehensive set of toolset provided by METAsoft. METAsoft features a comprehensive set of integrated, software quality tools including Test Coverage Analysis, Static Analysis & Metrics and Reverse Engineering & Regression Testing Support. Following is a summary overview of analysis capabilities available from METAsoft.

3.3.1 Dynamic Test Coverage Analysis

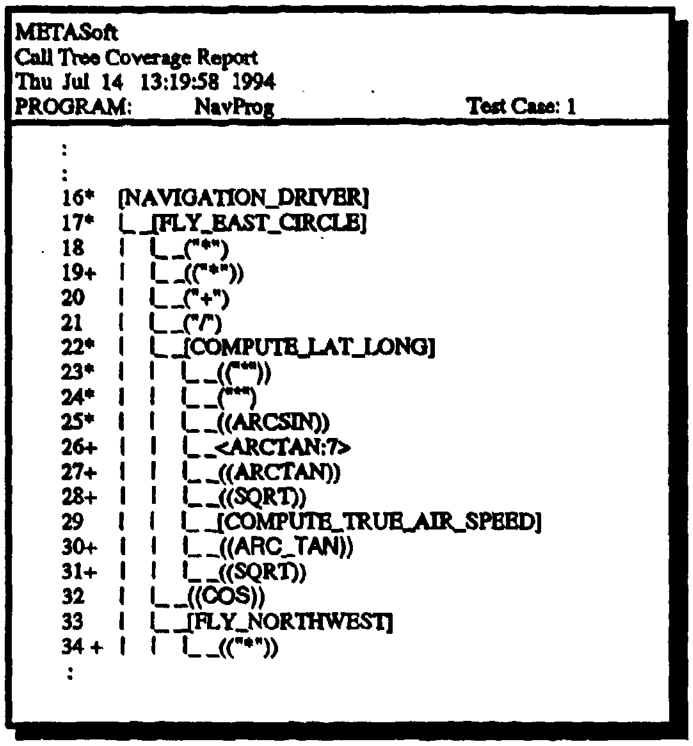
Software testing must ensure compliance with functional requirements. However, rigorous testing also requires that the test suite has adequately covered the source code. METAsoft offers complete support for test coverage analysis.

METAsoft provides both individual and cumulative test case coverage reports. It features color-coded flow charts that graphically depict the paths that were executed for a given test case. As additional test cases are executed statistics are accumulated to show cumulative test coverage over the entire program. The source code is accessed on-line and is also color-coded to illustrate test case coverage.

- Comprehensive Test Coverage
  - Unit and Cumulative Test Coverage Reports
  - Invocation Coverage
  - Statement Execution Coverage
  - Branch Execution Coverage
  - Subcondition Coverage
  - Management Oriented Summary Reports
- Control Flow Tracing

3.3.1.1 Call Tree Coverage Report

The Call Tree Coverage Report, illustrated below in Figure 3.3.1.1-1, indicates coverage of each node in the call tree using either an '*' or '+' immediately following the line number. An '*' indicates the node on the indicated line was not only called, but that it was called via that connecting arc. A '+' indicates that the node (i.e. the subroutine) on the indicated line was called during test execution, but not via that connecting arc.The report is generated simultaneously for the entire program and may be requested for either a unit test case or cumulative set of test cases.



```
MBTASoft
Call Tree Coverage Report
Thu Jul 14  13:19:58  1994
PROGRAM:        NavProg                       Test Case: 1

      :
      :
      16*   [NAVIGATION_DRIVER]
      17*   L_[FLY_EAST_CIRCLE]
      18    |  L_("*")
      19+   |  L_(("*"))
      20    |  L_("+")
      21    |  L_("/")
      22*   |  L_[COMPUTE_LAT_LONG]
      23*   |  |  L_(("**"))
      24*   |  |  L_("**")
      25*   |  |  L_((ARCSIN))
      26+   |  |  L_<ARCTAN:7>
      27+   |  |  L_((ARCTAN))
      28+   |  |  L_((SQRT))
      29    |  |  L_[COMPUTE_TRUE_AIR_SPEED]
      30+   |  |  L_((ARC_TAN))
      31+   |  |  L_((SQRT))
      32    |  L_((COS))
      33    |  L_[FLY_NORTHWEST]
      34 +  |  |  L_(("*"))
       :
```

Figure 3.3.1.1-Call Tree Coverage Report

3.3.1.2 Call Tree Coverage Diagram

The Call Tree Coverage Diagram, illustrated below in Figure 3.3.1.2-1, utilizes color applied to the Static Call Tree Diagram to indicate coverage. Color is applied to both the node icon and connecting arcs, indicating respectively whether the node was called at sometime during the test run or the call was made via that connection.

The diagram is generated simultaneously for the entire program and may be requested for either a unit test case or cumulative set of test cases.

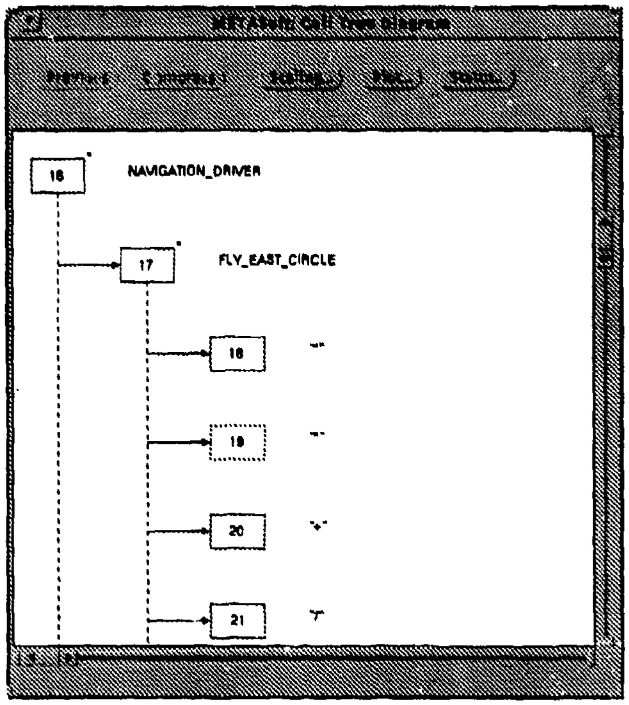
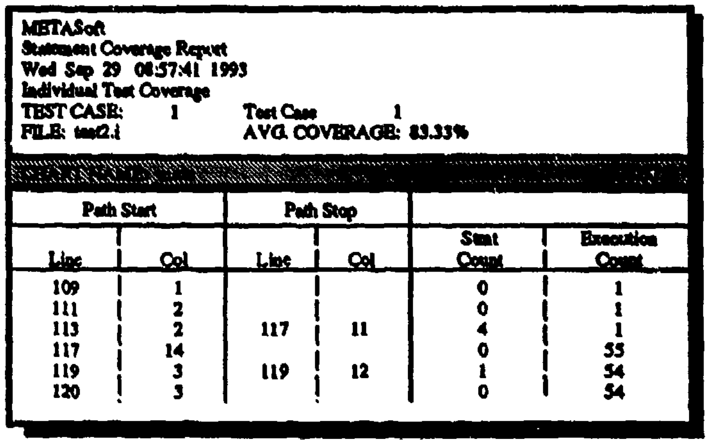


Figure 3.3.1.2-Call Tree Coverage Diagram

3.3.1.3 Statement Coverage Report

The Statement Coverage Report, illustrated below in Figure 3.3.1.3-1, indicates line and column boundary designations for each icon in the control flow diagram of the program. In cases where the icon is representative of a non-executable segment the path stop is assumed to be the end of line and the statement count is listed as zero. Otherwise, both a path start and stop are given and a statement count is shown. Additionally, the report shows the number of times that each statement group was executed.

The report is generated for all charts in the currently selected file. The average coverage for all statements in the file is shown in the report header. Coverage may be generated for either a unit test case or cumulative set of test cases.



METASoft
Statement Coverage Report
Wed Sep 29 08:57:41 1993
Individual Test Coverage
TEST CASE: 1 Test Case 1
FILE: test2.i AVG. COVERAGE: 83.33%

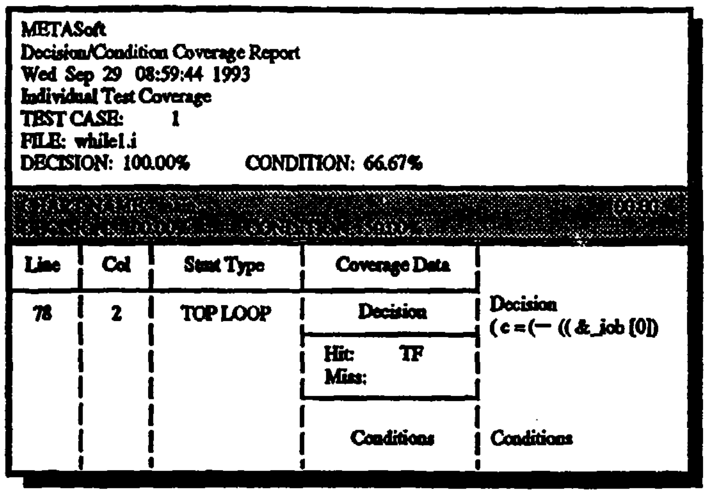
Path Start		Path Stop			
Line	Col	Line	Col	Stmt Count	Execution Count
109	1			0	1
111	2			0	1
113	2	117	11	4	1
117	14			0	55
119	3	119	12	1	54
120	3			0	54

Figure 3.3.1.3-1 Statement Coverage Report

3.3.1.4 Decision/Condition Coverage Reports

The Decision/Condition Combination Coverage Report, illustrated below in Figure 3.3.1.4-1, highlights each decision point in a file according to its chart location, line and column designation, and the type of decision. The decision text and a component condition breakdown are shown to the right of the coverage data. For each decision point hit and/or missed, information is specified for each possible outcome. Hit and/or miss information is also a given for each of the potential truth table combinations (2^n where there are n conditions) which determine the decision's outcome.

The report is generated for all charts in the currently selected file. The average decision and condition combination coverages are shown in the report header. Coverage may be generated for a unit test case or cumulative set of test cases.



```
METASoft
Decision/Condition Coverage Report
Wed Sep 29  08:59:44 1993
Individual Test Coverage
TEST CASE:      1
FILE: while1.i
DECISION: 100.00%       CONDITION: 66.67%
```

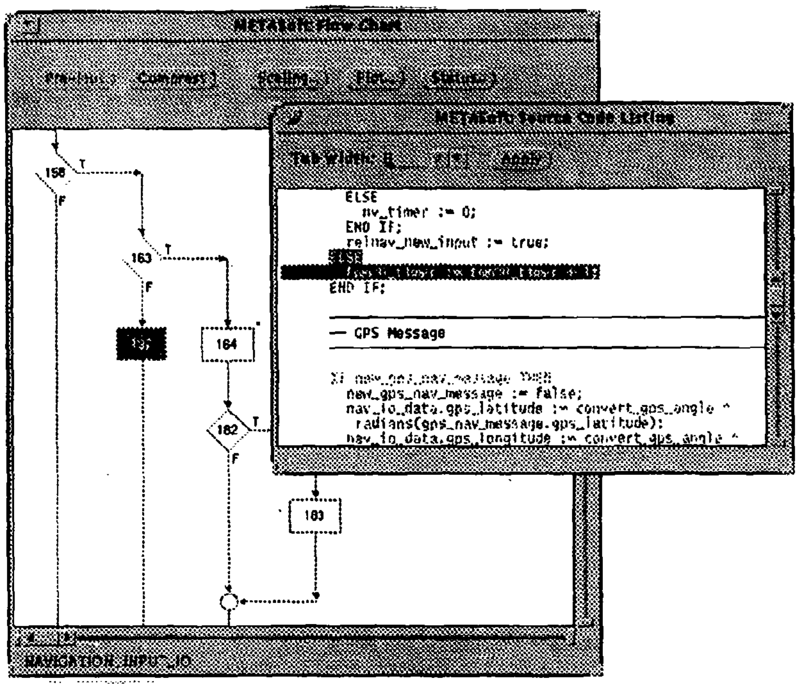
Line	Col	Stmt Type	Coverage Data	
78	2	TOP LOOP	Decision	Decision (c=(— ((&_job [0])
			Hit: TF Miss:	
			Conditions	Conditions

Figure 3.3.1.4-1 Decision/Condition Combination Coverage Report

3.3.1.5 Coverage Diagrams

Graphical representations of all coverage information collected is available in the form of coverage diagrams, as illustrated below in Figure 3.3.1.5-1. Coverage is represented as colors applied to the control flow diagram indicating full, partial and no coverage. For decision icons, the Decision/Condition Report information for the corresponding decision is available from a point and click menu. Cross-reference text is also color-coded to correspond to the coverage diagram and may be accessed directly from a point and click menu.

The diagram is generated for each chart (procedure/function/subprogram) in the program and may be requested for any unit test case or cumulative set of test cases.



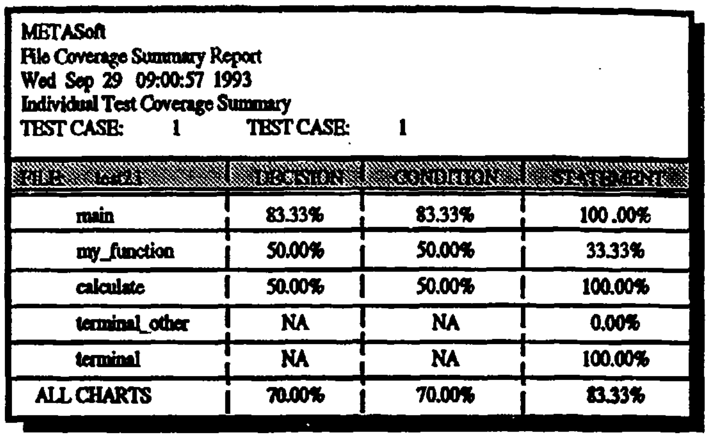
```
   ELSE
     nv_timer := 0;
   END IF;
   relnav_new_input := true;
ELSE
   [illegible]
END IF;

— GPS Message

IF new_gps_nav_message THEN
   new_gps_nav_message := false;
   nav_io_data.gps_latitude := convert_gps_angle ^
    radians(gps_nav_message.gps_latitude);
   nav_io_data.gps_longitude := convert_gps_angle ^
```

Figure 3.3.1.5-1 Coverage Diagram

3.3.1.6 Management Oriented Summary Reports

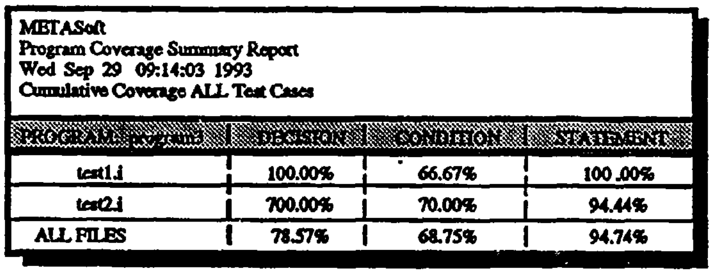
High-level overviews of all coverage information are available at both the file and program levels, illustrated below in Figure 3.3.1.6-1 and Figure 3.3.1.6-2. At the file level, a summary table of statement, decision and condition coverage averages for each chart (procedure/function/subprogram) in the file is available. At the program level, a summary table of statement, decision and condition coverage averages for each file in the program is available.

The summary reports may be generated for any unit test case or cumulative set of test cases.


METASoft
File Coverage Summary Report
Wed Sep 29 09:00:57 1993
Individual Test Coverage Summary
TEST CASE: 1 TEST CASE: 1

FILE: test2.i	DECISION	CONDITION	STATEMENT
main	83.33%	83.33%	100 .00%
my_function	50.00%	50.00%	33.33%
calculate	50.00%	50.00%	100.00%
terminal_other	NA	NA	0.00%
terminal	NA	NA	100.00%
ALL CHARTS	70.00%	70.00%	83.33%

Figure 3.3.1.6-1 File Coverage Summary Report


METASoft
Program Coverage Summary Report
Wed Sep 29 09:14:03 1993
Cumulative Coverage ALL Test Cases

PROGRAM: program3	DECISION	CONDITION	STATEMENT
test1.i	100.00%	66.67%	100 .00%
test2.i	700.00%	70.00%	94.44%
ALL FILES	78.57%	68.75%	94.74%

Figure 3.3.1.6-2 Program Coverage Report

3.3.2 Static Analysis Toolset

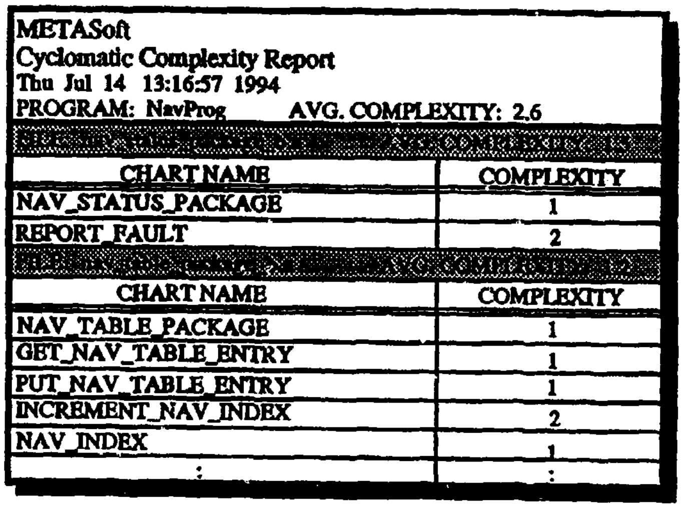
METAsoft provides a wide range of static analysis functionality to support increased reliability and quality in software. The Static Analysis utility detects errors early in the life cycle that could otherwise take days or weeks to find. METAsoft incorporates a wide range of software metrics to support the early identification of problem areas.

- Metrics
  - Cyclomatic Complexity
  - Selected Software Science metrics: Length, Vocabulary, Purity, Bugs, Volume
  - Code Statistics: SLOC, Comment

- Programming Standards Enforcement
- Identifier Usage Anomaly
- Identifier and Invocation Cross References
- Interval Analysis & Nesting Level
- Data Flow Analysis
- Call Tree Structure
- File/Package Interface Structure
- Flow Chart

3.3.2.1 Cyclomatic Complexity Report

The Cyclomatic Complexity Report, illustrated below in Figure 3.3.2.1-1, is derived from an implementation of the McCabe's Cyclomatic complexity algorithm. The report is generated simultaneously for the entire program and is segmented by file with Cyclomatic Complexity calculated for each function/procedure/subroutine contained in the file. Average complexities are generated at the file and program level.



METASoft
Cyclomatic Complexity Report
Thu Jul 14 13:16:57 1994
PROGRAM: NavProg AVG. COMPLEXITY: 2.6

[illegible]

CHART NAME	COMPLEXITY
NAV_STATUS_PACKAGE	1
REPORT_FAULT	2

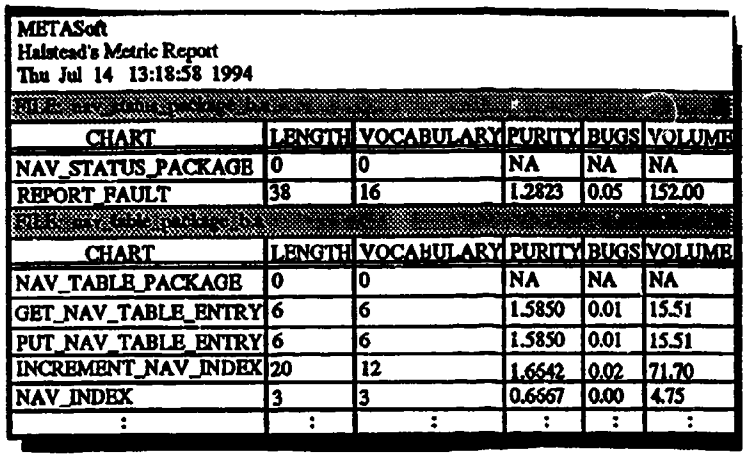
[illegible]

CHART NAME	COMPLEXITY
NAV_TABLE_PACKAGE	1
GET_NAV_TABLE_ENTRY	1
PUT_NAV_TABLE_ENTRY	1
INCREMENT_NAV_INDEX	2
NAV_INDEX	1
:	:

Figure 3.3.2.1-1 Cyclomatic Complexity Report

3.3.2.2 Halstead's Metrics Report

The Halstead's Metric Report, illustrated below in Figure 3.3.2.2-1, contains results of calculation of the five most widely recognized Halstead's Software Science Metrics (Length, Vocabulary, Purity, Bugs, and Volume). The report is generated simultaneously for the entire program and is segmented by file with the metric values being reported for each function/procedure/subroutine contained in each file.



METASoft
Halstead's Metric Report
Thu Jul 14 13:18:58 1994

[illegible]					
CHART	LENGTH	VOCABULARY	PURITY	BUGS	VOLUME
NAV_STATUS_PACKAGE	0	0	NA	NA	NA
REPORT_FAULT	38	16	1.2823	0.05	152.00
[illegible]					
CHART	LENGTH	VOCABULARY	PURITY	BUGS	VOLUME
NAV_TABLE_PACKAGE	0	0	NA	NA	NA
GET_NAV_TABLE_ENTRY	6	6	1.5850	0.01	15.51
PUT_NAV_TABLE_ENTRY	6	6	1.5850	0.01	15.51
INCREMENT_NAV_INDEX	20	12	1.6642	0.02	71.70
NAV_INDEX	3	3	0.6667	0.00	4.75
:	:	:	:	:	:

Figure 3.3.2.2-1 Halstead's Metric Report

3.3.2.3 Nesting Level Report

The Nesting Level Report, illustrated below in Figure 3.3.2.3-1, contains a metric of the maximum depth of logical nesting which occurs in each segment of the source code. The report is generated simultaneously for the entire program and is segmented by file with the metric values reported for each function/procedure/subroutine contained in each file.

METASoft
Nesting Level Report

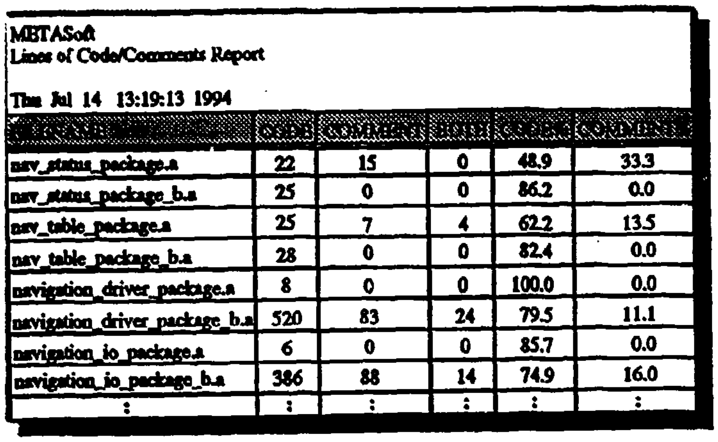
Thu Jul 14 13:19:22 1994

[illegible]	
CHART	**NESTING**
NAV_STATUS_PACKAGE	0
REPORT_FAULT	1
[illegible]	
CHART	**NESTING**
NAV_TABLE_PACKAGE	0
GET_NAV_TABLE_ENTRY	0
PUT_NAV_TABLE_ENTRY	0
INCREMENT_NAV_INDEX	1
NAV_INDEX	0
:	:

Figure 3.3.2.3-1 Nesting Level Report

3.3.2.4 SLOC/Comments Report

The Line Of Code/Comment Counts Report, illustrated below in Figure 3.3.2.4-1, contains a listing by file of the number of physical lines of code consisting of code only, comments only, and both code and comments. Also presented is a percentage of total lines in the file made up of code and comments. This report is generated simultaneously for the entire program and the statistics are segmented by file.


MBTASoft
Lines of Code/Comments Report

Thu Jul 14 13:19:13 1994

FILENAME	CODE	COMMENT	BOTH	%CODE	%COMMENT
nav_status_package.a	22	15	0	48.9	33.3
nav_status_package_b.a	25	0	0	86.2	0.0
nav_table_package.a	25	7	4	62.2	13.5
nav_table_package_b.a	28	0	0	82.4	0.0
navigation_driver_package.a	8	0	0	100.0	0.0
navigation_driver_package_b.a	520	83	24	79.5	11.1
navigation_io_package.a	6	0	0	85.7	0.0
navigation_io_package_b.a	386	88	14	74.9	16.0
:	:	:	:	:	:

Figure 3.3.2.4-1 Line Of Code/Comment Counts Report

3.3.2.5 Identifier Usage Anomaly Report

The Identifier Usage Anomaly Report, illustrated below in Figure 3.3.2.5-1, indicates identifier oriented potential errors including identifiers that are:

- declared and never used,
- used and not explicitly declared,
- declared and initialized but never used,
- declared and used but never explicitly initialized
- declared and used but conditionally initialized.

Results are reported simultaneously for the entire program and are segmented by category of fault with types, constants, function/procedure names, and variables separately segmented.

METASoft Static Analysis Report - Usage Anomalies Report Thu Jul 14 13:18:41 1994 PROGRAM: NavProg	
d = defined, s = set, r = read, u = used, c = called	
TYPES NEVER USED	
CHART: NAV_STATUS_PACKAGE.NAV_FAULT_STACK	
TYPE: LOCATION:	FAULT_PROCESSING_ARRAY_TYPE nav_status_package.a 39(d)
TYPE: LOCATION:	FAULT_PROCESSING_RECORD nav_status_package.a 39(d)
:	:
CONSTANTS NEVER USED	
CHART: DISPLAY_GLOBAL_DATA	
CONSTANT: LOCATION:	IFPM_ALERT display_global_data.a 8(d)
CONSTANT: LOCATION:	NAV_STATUS display_global_data.a 6(d)
:	:
VARIABLES NEVER USED	
CHART: DRIVER_TO_NAV_DRIVER	
VARIABLE: LOCATION:	HEADING_MODE driver_to_nav_driver.a 6(d)
:	:
VARIABLES NEVER SET	
CHART: DRIVER_TO_NAV_DRIVER	
VARIABLE: LOCATION:	ADC_ACTIVE driver_to_nav_driver.a 17(d) navigation_driver_package_b.a 363(r)
VARIABLE: LOCATION:	GPS_ACTIVE driver_to_nav_driver.a 16(d) navigation_driver_package_b.a 335(r)
:	:
VARIABLES NEVER READ	
CHART: NAVIGATION_DRIVER_PACKAGE	
VARIABLE: LOCATION:	ALPHA navigation_driver_package_b.a 113(d) 236(s) 529(s) 586(s)
:	:
CHARTS NEVER CALLED	
CHART: NAVIGATION_DRIVER_PACKAGE	
CHART: LOCATION:	"*" navigation_driver_package_b.a 182(d)
CHART: LOCATION:	INITIALIZE navigation_driver_package_.a 5(d)
:	:

Figure 3.3.2.5-1 Identifier Usage Anomaly Report

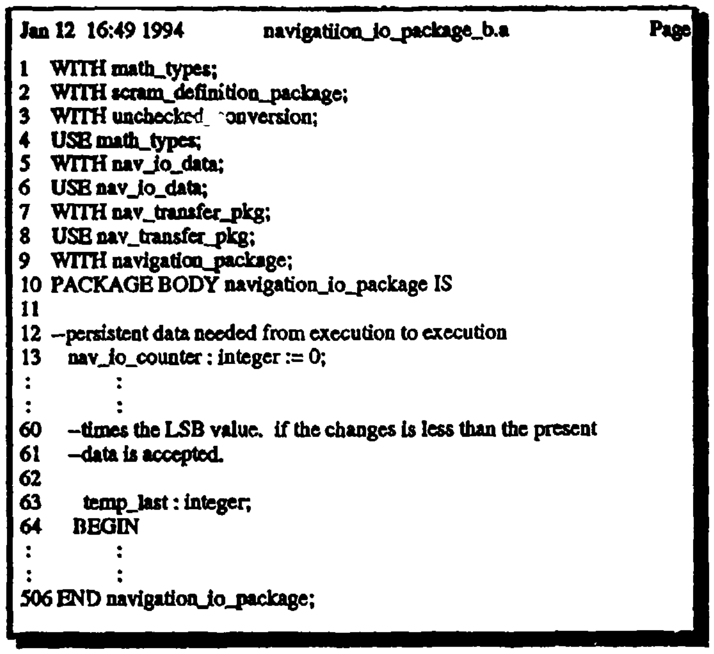
3.3.3 Reverse Engineering Support

METAsoft provides a wide range of reports and documentation to support reverse engineering of software systems. Report formats include control flow diagrams at the module, call tree, and individual function/procedure level. Module interface diagrams are presented in a modified Booch notation indicating both the physical nesting of modules as well as the logical interaction of their data elements. Call tree diagrams are presented in both textual form and as diagrams.

- Source Code Cross-Reference Listing
- Control Flow Diagrams
- Identifier Cross-Reference Listing
- Module Interface Diagrams
- Call Tree Report
- Call Tree Diagram
- Data Flow Report
- Calling Charts Report
- Called Charts Report

3.3.3.1 Source Code Cross-Reference Listing

Source Code Cross-Reference Listings, illustrated below in Figure 3.3.3.1-1, are simply copies of the user's original source with line numbers prefaced to each code line for ease in use as a cross-reference. The line numbers contained in this listing match exactly with all of those utilized throughout METAsoft reports.

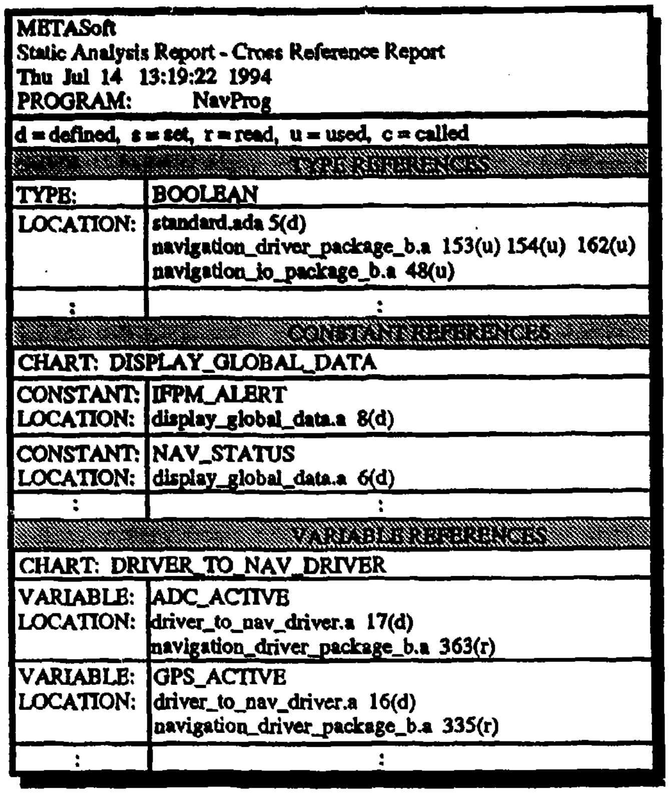


```
Jan 12  16:49 1994        navigatiion_io_package_b.a              Page
1   WITH math_types;
2   WITH scram_definition_package;
3   WITH unchecked_ conversion;
4   USE math_types;
5   WITH nav_io_data;
6   USE nav_io_data;
7   WITH nav_transfer_pkg;
8   USE nav_transfer_pkg;
9   WITH navigation_package;
10  PACKAGE BODY navigation_io_package IS
11
12  --persistent data needed from execution to execution
13    nav_io_counter : integer := 0;
:         :
:         :
60    --times the LSB value.  if the changes is less than the present
61    --data is accepted.
62
63      temp_last : integer;
64     BEGIN
:         :
:         :
506 END navigation_io_package;
```

Figure 3.3.3.1-1 Source Code Cross-Reference Listings

3.3.3.2 Identifier Cross-Reference Listing

Complete cross-reference listings, illustrated below in Figure 3.3.3.2-1, are provided for all identifiers contained in a segment of processed code. The identifier listings are segmented by category (type, constant and variable) with each identifier highlighted according to the file and line number where it occurs and the method in which it is being referenced (definition, set, read, used, called). This report encompasses all identifiers in the entire program.



METASoft Static Analysis Report - Cross Reference Report Thu Jul 14 13:19:22 1994 PROGRAM: NavProg	
d = defined, s = set, r = read, u = used, c = called	
	TYPE REFERENCES
TYPE:	BOOLEAN
LOCATION:	standard.ada 5(d) navigation_driver_package_b.a 153(u) 154(u) 162(u) navigation_io_package_b.a 48(u)
:	:
	CONSTANT REFERENCES
CHART: DISPLAY_GLOBAL_DATA	
CONSTANT: LOCATION:	IFPM_ALERT display_global_data.a 8(d)
CONSTANT: LOCATION:	NAV_STATUS display_global_data.a 6(d)
:	:
	VARIABLE REFERENCES
CHART: DRIVER_TO_NAV_DRIVER	
VARIABLE: LOCATION:	ADC_ACTIVE driver_to_nav_driver.a 17(d) navigation_driver_package_b.a 363(r)
VARIABLE: LOCATION:	GPS_ACTIVE driver_to_nav_driver.a 16(d) navigation_driver_package_b.a 335(r)
:	:

Figure 3.3.3.2-1 Identifier Cross Reference Listings

3.3.3.3 Module Interface Diagrams

Module Interface Diagrams, illustrated below in Figure 3.3.3.3-1, display the hierarchy of modules contained within a program. The diagram used a Booch style notation (open and closed connector boxes) to identify physical inclusion versus references to one module from within another. The diagram is presented for the entire program and can be printed via postscript tiling across multiple 8 1/2" x 11" pages. Numeric references contained within blocks are intended as an index.

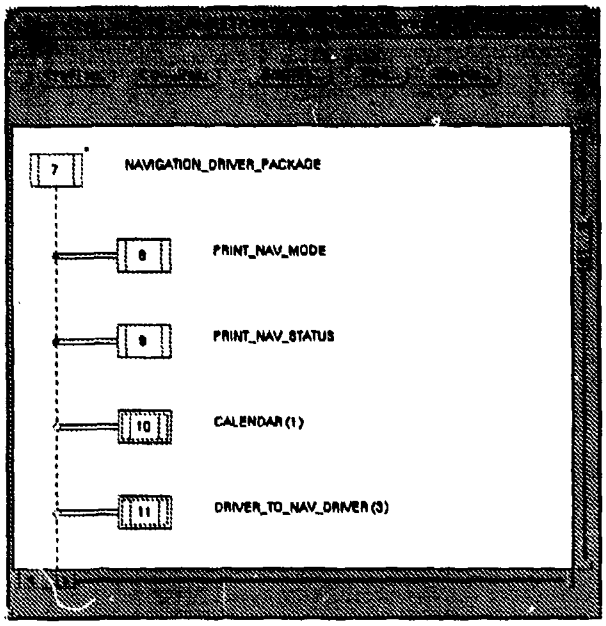
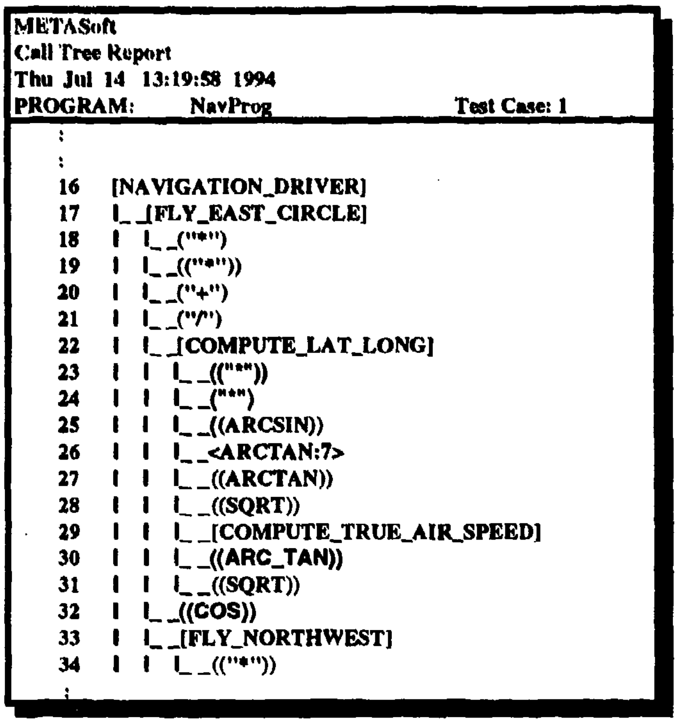


Figure 3.3.3.3-1 Module Interface Diagrams

3.3.3.4 Call Tree Report

The Call Tree Report, illustrated below in Figure 3.3.3.4-1, identifies the complete invocation structure for all source code processed within a given program. The structure uses an indentation scheme to represent each successive level of control logic nesting. Nomenclature ([], (), (()), <>) represents the varying types of nodes in the tree including respectively:

- parent in which other calls are made,
- terminal call for which all source has been processed,
- terminal call for which no source has been processed,
- repeated node with reference to its original occurrence.



```
METASoft
Call Tree Report
Thu Jul 14  13:19:58  1994
PROGRAM:       NavProg                        Test Case: 1
     :
     :
     16    [NAVIGATION_DRIVER]
     17    |_ _[FLY_EAST_CIRCLE]
     18    |   |_ _("*")
     19    |   |_ _(("*"))
     20    |   |_ _("+")
     21    |   |_ _("/")
     22    |   |_ _[COMPUTE_LAT_LONG]
     23    |   |   |_ _(("*"))
     24    |   |   |_ _("*")
     25    |   |   |_ _((ARCSIN))
     26    |   |   |_ _<ARCTAN:7>
     27    |   |   |_ _((ARCTAN))
     28    |   |   |_ _((SQRT))
     29    |   |   |_ _[COMPUTE_TRUE_AIR_SPEED]
     30    |   |   |_ _((ARC_TAN))
     31    |   |   |_ _((SQRT))
     32    |   |_ _((COS))
     33    |   |_ _[FLY_NORTHWEST]
     34    |   |   |_ _(("*"))
     :
```

Figure 3.3.3.4-1 Call Tree Report

3.3.3.5 Call Tree Diagram

The Call Tree Diagram, illustrated below in Figure 3.3.3.5-1, displays an alternative representation of the same information contained in a Call Tree Report. The information is presented as a series of icons representing each of the calling categories (parent non-terminal, terminal source, terminal no source, and repeated node). In an interactive mode METAsoft provides detailed flow diagrams of the source for each call, available directly from the call tree diagram in a point and click menu fashion. The Call Tree Diagram is generated and displayed at the program level.

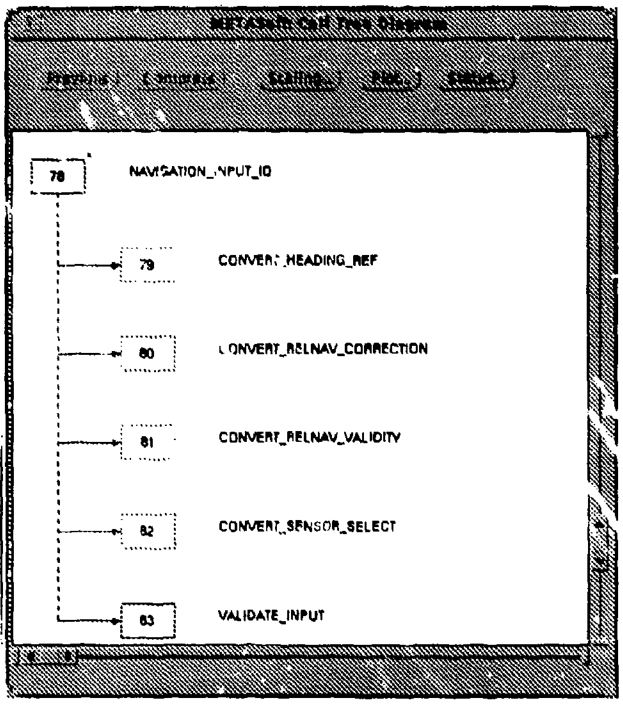


Figure 3.3.3.5-1 Call Tree Diagram

3.3.3.6 Control Flow Diagram

A Control Flow Diagram, illustrated below in Figure 3.3.3.6-1, is generated for each independent code segment (procedure, function, subroutine, etc.). These diagrams are represented in the form of flow charts of the control logic of the underlying source code. Rectangles represent contiguous code blocks for which there is no exit or branch point. Diamonds represent logic branch points. Hexagons represent multi-way decisions such as a case construct. Ovals represent the beginning and ending of scoping blocks. Small circles are used for graph vertices and rejoin points. The numbers contained within each icon represent the beginning line number of the corresponding block of source from the user's original files. These numbers correspond directly to the source line numbers generated in the Source Code Xref Report. When utilizing METAsoft's interactive interface this source is available directly through a point and click menu with the source corresponding to the selected graphical icon highlighted.

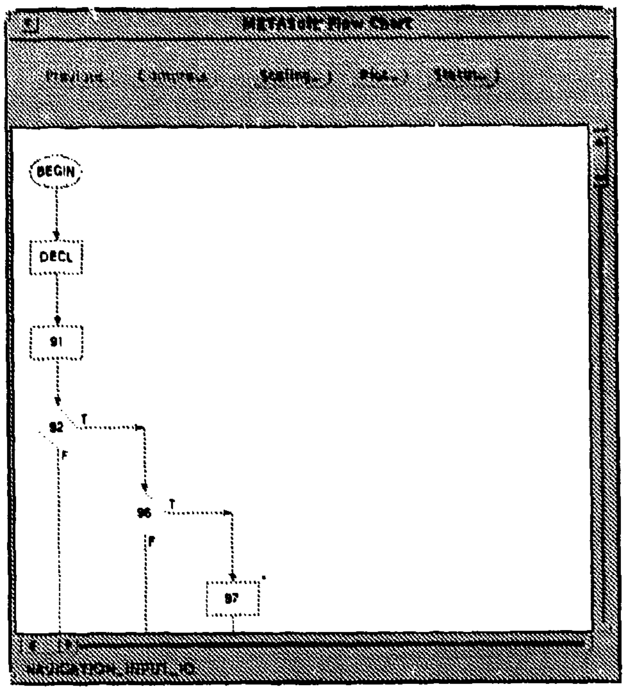
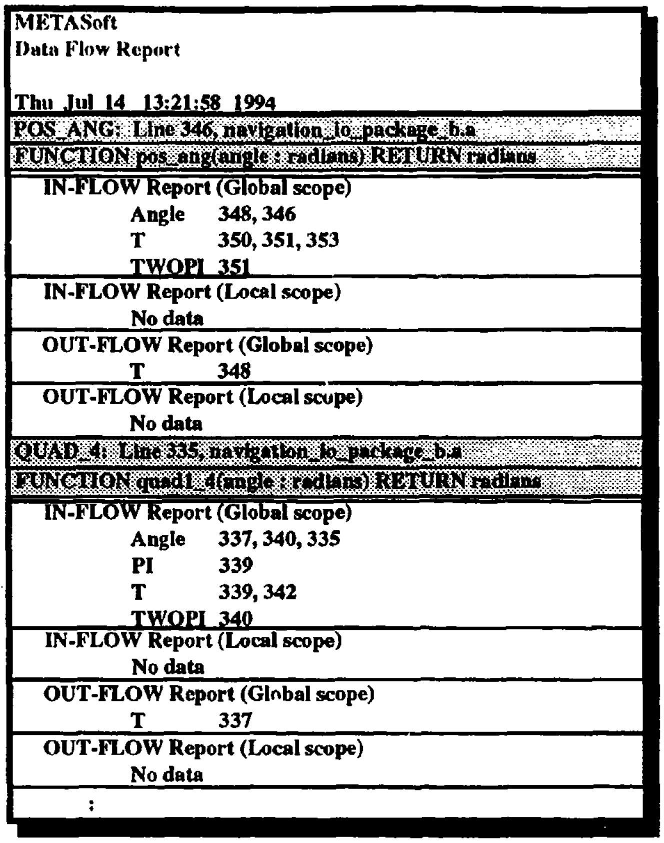


Figure 3.3.3.6-1 Control Flow Diagram

3.3.3.7 Data Flow Report

The Data Flow Report, illustrated below in Figure 3.3.3.7-1, outlines all program interfaces (function, procedure, package, file, etc.) and indicates both local and global data flow across that interface. The report may be requested for the contents of a single file or for the entire program. References are given to the declaration of the program segment (function, procedure, subprogram, package etc.) along with the text of the formal definition of the interface. Each piece of data crossing the interface of that program segment is listed in the proper in/out-global/local category along with the line number from the source file on which the actual interface occurred.



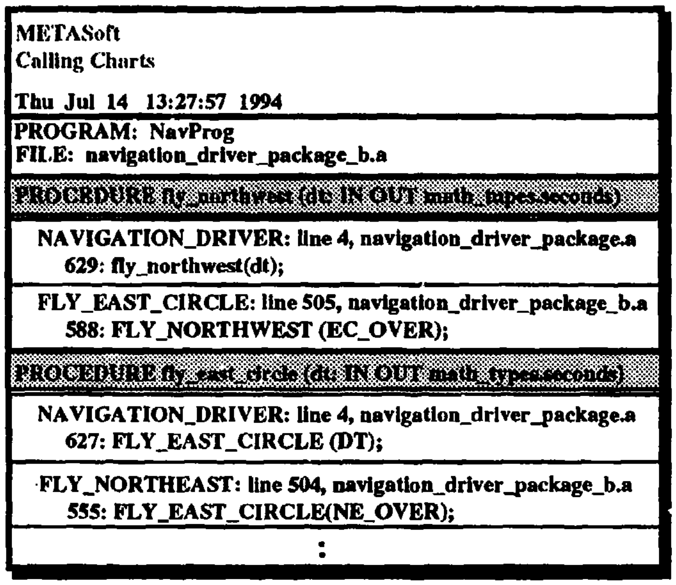
```
METASoft
Data Flow Report

Thu Jul 14 13:21:58 1994
POS_ANG: Line 346, navigation_io_package_b.a
FUNCTION pos_ang(angle : radians) RETURN radians
   IN-FLOW Report (Global scope)
             Angle     348, 346
             T         350, 351, 353
             TWOPI     351
   IN-FLOW Report (Local scope)
             No data
   OUT-FLOW Report (Global scope)
             T         348
   OUT-FLOW Report (Local scope)
             No data
QUAD_4: Line 335, navigation_io_package_b.a
FUNCTION quad1_4(angle : radians) RETURN radians
   IN-FLOW Report (Global scope)
             Angle     337, 340, 335
             PI        339
             T         339, 342
             TWOPI     340
   IN-FLOW Report (Local scope)
             No data
   OUT-FLOW Report (Global scope)
             T         337
   OUT-FLOW Report (Local scope)
             No data
         :
```

Figure 3.3.3.7-1 Data Flow Report

;

3.3.3.8 Calling Charts Report

The Calling Charts Report, illustrated below in Figure 3.3.3.8-1, may be generated for only those procedures, functions, and/or subroutines contained in a particular file, or fro all of them in an entire program. The report is segmented by file specification with each code unit shown along with its full declaration and position of declaration in that file. Following this information is a list of all calls made to that code unit with the file and line number location of the call and the text of the call complete with actual parameters.

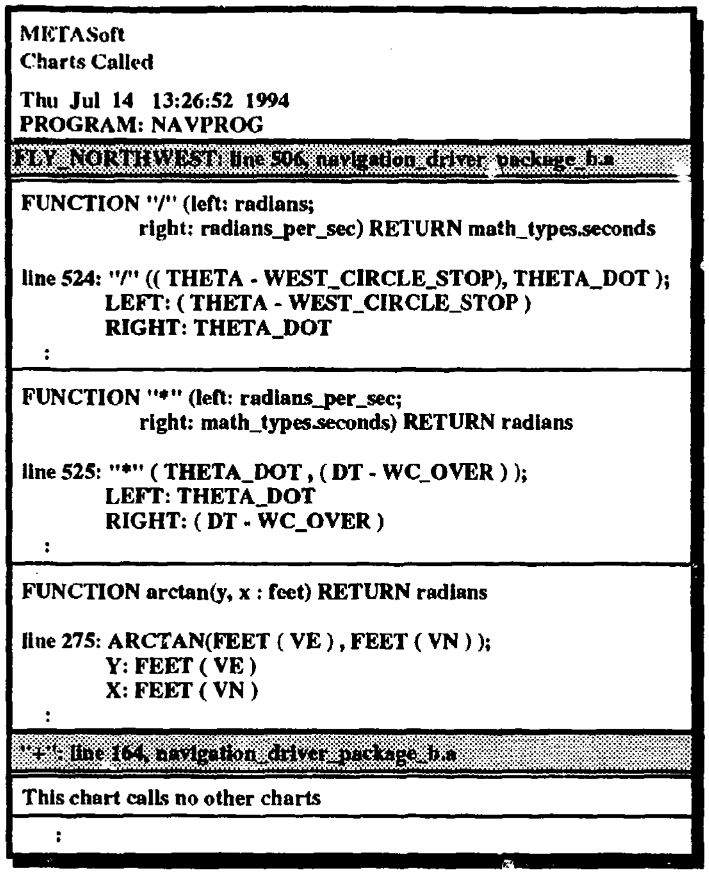


```
METASoft
Calling Charts
Thu Jul 14  13:27:57  1994
PROGRAM:  NavProg
FILE:  navigation_driver_package_b.a
PROCEDURE fly_northwest (dt: IN OUT math_types.seconds)
   NAVIGATION_DRIVER: line 4, navigation_driver_package.a
      629: fly_northwest(dt);
   FLY_EAST_CIRCLE: line 505, navigation_driver_package_b.a
      588: FLY_NORTHWEST (EC_OVER);
PROCEDURE fly_east_circle (dt: IN OUT math_types.seconds)
   NAVIGATION_DRIVER: line 4, navigation_driver_package.a
      627: FLY_EAST_CIRCLE (DT);
   FLY_NORTHEAST: line 504, navigation_driver_package_b.a
      555: FLY_EAST_CIRCLE(NE_OVER);
                            :
```

Figure 3.3.3.8-1 Calling Charts Report

3.3.3.9 Called Charts Report

The Called Charts Report, illustrated below in Figure 3.3.3.9-1, may be generated for only those procedures, functions, and/or subroutines contained in a particular file or for all of them in an entire program. Each function, procedure, and/or subroutine is listed along with its line number and file location. Following this information is a list of all calls made within that function, complete with the formal declaration of the function as well as the line number location and all actual parameters for each occurrence of the call within the file.



```
METASoft
Charts Called

Thu Jul 14  13:26:52  1994
PROGRAM: NAVPROG
FLY_NORTHWEST: line 506, navigation_driver_package_b.a
FUNCTION "/" (left: radians;
              right: radians_per_sec) RETURN math_types.seconds

line 524: "/" (( THETA - WEST_CIRCLE_STOP), THETA_DOT );
          LEFT: ( THETA - WEST_CIRCLE_STOP )
          RIGHT: THETA_DOT
   :
FUNCTION "*" (left: radians_per_sec;
              right: math_types.seconds) RETURN radians

line 525: "*" ( THETA_DOT , ( DT - WC_OVER ) );
          LEFT: THETA_DOT
          RIGHT: ( DT - WC_OVER )
   :
FUNCTION arctan(y, x : feet) RETURN radians

line 275: ARCTAN(FEET ( VE ) , FEET ( VN ) );
          Y: FEET ( VE )
          X: FEET ( VN )
   :
"+": line 164, navigation_driver_package_b.a
This chart calls no other charts
    :
```

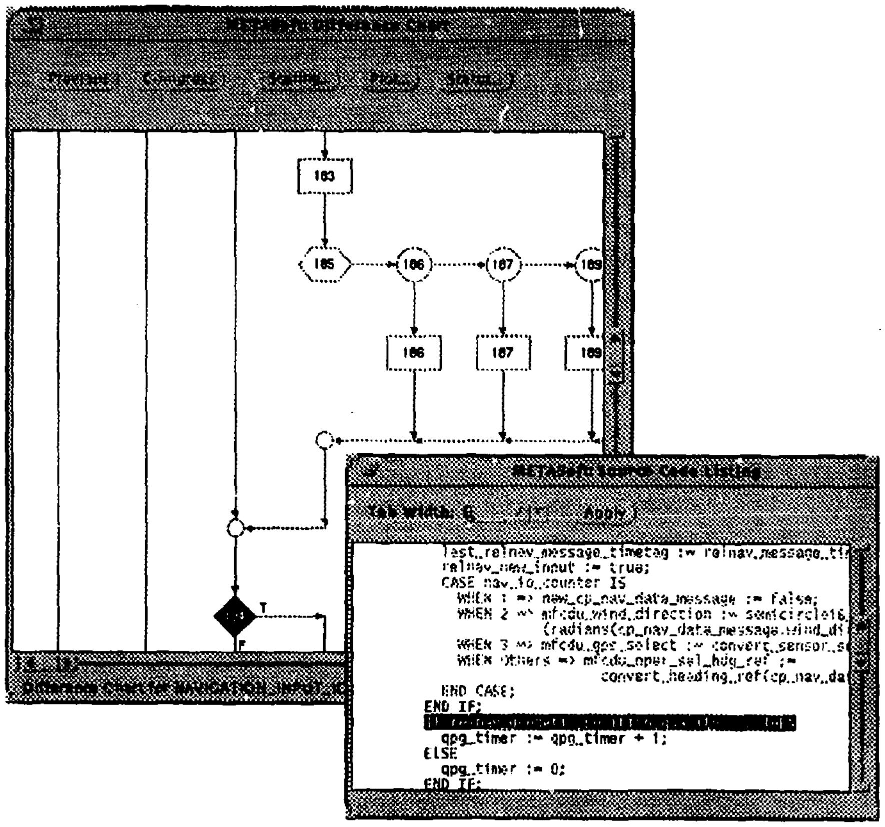
Figure 3.3.3.9-1 Called Charts Report

3.3.4 Regression Testing Support

METAsoft incorporates a sophisticated differencing mechanism to highlight changes between two versions of source code and to show the areas of source potentially affected by those changes. Control Flow diagrams are color coded to indicate areas that have not changed, segments of code that have been inserted, segments that have been deleted, segments with textual changes that do not affect control flow, and areas of the code potentially affected by any of these changes. By analyzing these diagrams the regression test suite can be targeted to focus testing effort on the areas affected by source changes.

3.3.4.1 Source Code Configuration Difference Diagrams

The Source Code Configuration Difference Diagram, illustrated below in Figure 3.3.4.1-1, is utilized to highlight changes between two versions of source code and to show the areas of the source which are potentially affected by those changes. The chart mechanism is the same as in the control flow diagram. Color is applied to indicate areas which have not changed, segments of code which have inserted or deleted, and segments or code which contain a textual change with no change to control flow of the chart. As with the control flow diagram, each icon contains the beginning line number in the original source file of the segment of code which corresponds to the text associated with the icon. All source text, except for deleted segments which correspond to the older file version, can be viewed via the interactive METAsoft interface and are representative of the source text contained in the more recent version of the two files being compared.



```
    last_relnav_message_timetag := relnav_message_ti
    relnav_new_input := true;
    CASE nav_io_counter IS
      WHEN 1 => new_cp_nav_data_message := false;
      WHEN 2 => mfcdu_wind_direction := semicircle16
                   (radians(cp_nav_data_message.wind_di
      WHEN 3 => mfcdu_gps_select := convert_sensor_s
      WHEN Others => mfcdu_oper_sel_hdg_ref :=
                            convert_heading_ref(cp_nav_da
    END CASE;
  END IF;
  [illegible]
    qpg_timer := qpg_timer + 1;
  ELSE
    qpg_timer := 0;
  END IF;
```

Figure 3.3.4.1-1 Source Code Configuration Difference Diagram

4.0 Distributed Software Background

As advanced computer technology becomes less costly and multiprocessing environments more common, the phrase "parallel and distributed computing" is heard with increasing frequency. However, defining this term is not so simple. Traditionally the term concurrent computing has been used to represent a system where independent processes execute simultaneously with others. Originally these processes were implemented on multiprogramming systems, where there is one CPU shared among processes. Parallel computing gained popularity when multiprocessor architectures brought about the capacity to execute processes simultaneously using several processing units.

In general, parallel computing architectures fall into one of several classes: SIMD, SPMD, shared memory MIMD and distributed memory MIMD. Each class is defined in the following paragraphs.

A SIMD (Single Instruction Stream / Multiple Data Stream) system is characterized by the fact that each processor executes the same set of instructions in lock-step on different sets of data. That is, each process is executing the same instruction at the same time. Examples of SIMD systems are those that execute massively-parallel, or fine-grained parallel code, such as an array processor.

An SPMD (Single Program / Multiple Data Stream) system is often considered an extension of the SIMD classification or a restriction of the MIMD classification (see below). Generally SPMD programs execute a set of identical processes. Each process progresses at its own rate. An example of an SPMD program might be characterized by the use of fork/join procedures or do_across loops. Several identical copies of a process are spawned, and upon completion the control flow joins back together at the parent process.

MIMD (Multiple Instruction Stream / Multiple Data Stream) is an architecture in which many instruction streams execute in parallel on multiple data sets. Each process may be unique and proceeds at its own rate. Some MIMD architectures, such as the Intel Hypercube, contain a set of identical processors, others may have non-identical processors.

MIMD systems can also be subclassified by the memory management paradigm. Closely coupled architectures may share memory among processors, while in other systems each processor has its own private memory and none is shared. The latter environment is referred to as distributed memory. In a distributed memory system, the only manner in which processes on separate processors can communicate with one another is via message passing. An example of a non-homogeneous, distributed memory multiprocessing environment is a set of networked computers.

Distributed computing is a special form of parallel computing. Often it is defined as loosely-coupled parallel computation, characterized by the use of communicating processes on separate heterogeneous computer systems. Thus, distributed computing is most appropriately categorized as distributed memory MIMD computing. Note that this indicates that processes on different processors communicate by message passing only, but processes on the same processor may use either message passing or shared memory within their system for communication.

4.1 Programming distributed systems

Along with hardware support for distributed systems, there must be software support for programming such systems as well. Software support generally falls into two broad categories: high-level languages designed explicitly for supporting distributed programming or language extensions in the form of a library of support routines for distributed processing.

Many languages have been developed expressly to provide support for concurrent programming. A most notable example is Ada which also supports real-time environments. Ada tasking features allow the creation of independently executing processes or tasks. Tasks communicate by sending messages using the rendezvous mechanism. However, Ada alone is not sufficient for distributed programming. Compilers and support software capable of allocating tasks to separate processors and managing the communications between them are not readily available. Distributed systems written in Ada are generally composed of Ada programs in each processor that use support routines for inter-processor communication. Another concurrent language is CSP (communicating sequential processes). Like Ada, CSP supports synchronized point-to-point communication between processes. In general, however, most languages designed to support distributed computing at a high-level are not available across a wide variety of machines.

To facilitate distributed programming in a MIMD environment, many manufacturers of multiprocessor systems provide a set of support routines that can be used by traditional programming languages such as Fortran, and C (sometimes Ada). These support routines generally provide mechanisms for interprocess communication via message passing, as well as the support needed to create and load processes on specific processors. The support libraries hide the details of the underlying communication details from the programmer, who is still able to program in a traditional language with which they are already familiar. Any tasking activities such as synchronization or data sharing takes place using either synchronous or asynchronous message passing, depending on the requirements of the program. An example of such an environment is Intel's Hypercube series of machines. Support libraries are provided for both Fortran and C.

4.2 General characteristics of interprocess communication

As mentioned previously, Ada processes communicate through the rendezvous. One task calls another's entry point. The input and output parameters to the entry call define the message(s) to be passed at the beginning and/or end of the rendezvous. The rendezvous is an important process synchronization mechanism in Ada. That is, a task requesting a rendezvous must wait for the destination task to be ready to accept the rendezvous. Once a rendezvous is initiated, the calling task must wait until the rendezvous completes. At the accepting end of a rendezvous, once a task has executed the accept statement, it must wait until another task requests an entry to that accept to proceed. Of course there are other more sophisticated tasking mechanisms in Ada such as the select statement, but this provides an overview of the synchronous behavior of process interactions.

CSP, like Ada, uses similar synchronization between communicating processes. The main difference is that in Ada, a task may accept an entry call from any other task, while in CSP communication takes place over a dedicated point-to-point channel. For the case where it is necessary to accept communications from one of a set of calling tasks, CSP has a statement similar to the select that non-deterministically chooses the next channel from which it will accept communication.

A feature of these highly synchronous, named communication pathways, is that it is often possible to statically determine which tasks may be involved in a particular synchronization event (such as an accept). Also, many software testing techniques existing for concurrent programs are based on the Ada model of process synchronization.

Interprocess communication in the case of communications support libraries usually behaves in a much different manner. Communication takes place when one process prepares a message buffer and sends it to a destination process or group of processes. The data in the buffer is usually merely a stream of bytes. It is up to the receiver to decipher the contents of the buffer.

A send routine may be blocking or nonblocking. That is, if the send routine is blocking, the sending process does not resume execution until the buffer reaches its destination. If it is nonblocking, the send routine dispatches the message, then returns control to the sender. Often this definition of send is referred to as an asynchronous send. The receive function may also be synchronous or asynchronous. In the first case, when a receive function is called, the receiving process waits until the requested buffer arrives. In the latter case, the receive function returns the message if it is there, otherwise it returns a value indicating the message has not yet arrived. If more than one message arrives for a process before it is ready to receive them, they are generally maintained in a queue by the underlying communication subsystem.

The send routine usually takes as one of its parameters a single destination process identifier or a special identifier indicating the same message is to be broadcast to a group of processes. Similarly, the receive routine contains a parameter used to identify from which process the next message is to be retrieved. This parameter may indicate that the message may come from any sender.

The combination of asynchronous communication capabilities with the use of subroutine parameters rather than names to identify sources and destinations can make it very difficult to statically detect all of the potential process communication pathways in a system. Clearly, this mechanism is very different from the well-named nature of Ada tasking.

4.3 General purpose message-passing programming support

In an environment composed of a collection of heterogeneous processors such as a networked set of computers, programmers sometimes must resort to using network communication functions to perform inter-process activities, at least between processes on different machines. This can be very difficult since different computers may have different network interface routines.

In recognition of the need for distributed programming support, the development of PVM (Parallel Virtual Machine) started in 1989 at Oak Ridge National Laboratory. The software for PVM is distributed free of charge and is being used for computational applications worldwide. Interest in PVM as a research and application support tool is evidenced by the existence of a USENET newsgroup dedicated to the discussion of PVM and related issues.

PVM provides facilities for a heterogeneous network of parallel and/or serial computers to work as a single distributed memory parallel computer. Application programs view PVM as a general, portable and flexible resource that supports programs that fit into the message passing model of computation. PVM handles data conversions between architectures and allows the computers that make up the virtual machine to be connected by a variety of networks and communications protocols. As a result, PVM also supports networks of computers connected by the Internet, thus permitting truly world-wide computations.

PVM is implemented in two parts. A daemon process runs on each of the computers that make up the virtual machine. The second part is a library of interface routines that contain support functions for message passing, process creation and task coordination. Currently, there are support interfaces for both Fortran and C. Although the support library creates another level of software between the user program and the underlying communications mechanisms, this approach greatly simplifies distributed computing involving several distinct computer architectures for which no uniform distributed operating system exists.

PVM is implemented on over 20 platforms an ongoing research supports implementing PVM on other systems. The table below lists the supported architectures for PVM 3.3 as of June 1994.

Table 1:

CPU ID	Architecture
AFX8	Alliant FX/8
ALPHA	DEC Alpha/OSF-1
ALPHAMP	DEC Alpha multiprocessor/OSF >= 3.0
BAL	Sequent Balance
BFLY	BBN Butterfly TC2000
BSD386	80[34]86 running BSDI, 386BSD, NetBSD, FreeBSD
CM2	Thinking Machines CM-2 Sun front
CM5	Thinking Machines CM-5
CNVX	Convex using IEEE floating-point
CNVXN	Convex using native f.p.
CRAY	Cray
CRAY2	Cray-2
CRAYSMP	Cray S-MP
DGAV	Data General Aviion
E88K	Encore 88000
HP300	HP 9000 68000 cpu
HPPA	HP 9000 PA-Risc
I860	Intel RX Hypercube [3]
IPSC2	Intel IPSC/2
KSR1	Kendall Square
LINUX	80[34]86 running Linux
MASPAR	Maspar/Dec Mips front-end

Table 1:

CPU ID	Architecture
MIPS	Mips
NEXT	NeXT
PGON	Intel Paragon
PMAX	DEC/Mips arch (3100, 5000, etc.)
POWER4	IBM Power-4
RS6K	IBM/RS6000
RT	IBM/RT
SGI	Silicon Graphics IRIS
SGI5	Silicon Graphics IRIS OS >= 5.0
SGIMP	Silicon Graphics IRIS multiprocessor
SUN3	Sun 3
SUN4	Sun 4, 4c, sparc, etc.
SUN4SOL2	Sun 4 running Solaris
SUNMP	Sun 4 multiprocessor
SYMM	Sequent Symmetry
TITN	Stardent Titan
UVAX	DEC/Microvax
VCM2	Thinking Machines CM-2 Vax front end

Because PVM incorporates fundamental message communication activities found in many vendor specific support libraries, it is important to note that software testing methods that are applicable to PVM can also be applied to most distributed memory/message-passing environments.

4.4 Standardization of message passing

The May 1994 report *MPI: A Message-Passing Interface Standard*, states that the Message Passing Interface Forum (MPIF) includes participation from more than 40 organizations. The MPIF has been meeting since November 1992 to develop a uniform set of message passing facilities in the form of an interface to a standard library of communications services. The goal of the Message Passing Interface is to develop a standard for writing programs using the message passing paradigm. Supporting goals are to make the MPI standard: portable, efficient, and flexible.

The message passing paradigm is currently in wide use on variety of parallel machines and is particularly useful for those architectures having distributed memory.

MPI includes support for most common interprocess activities including both blocking and non-blocking communications, broadcast communications and barriers. MPI has many communications features similar to those found in PVM and other message-passing support environments. In general, MPI includes those features found in PVM plus additional refinements.

4.5 Evaluating Testing Strategies for Distributed Programs

The preceding discussion illustrates the interest in and development of programming support environments capable of supporting distributed systems. PVM and MPI are examples of environments supporting distributed programs and are fairly representative examples of the types of languages/language extensions that the ETADS tool must be capable of handling.

In order to fully appreciate the trades encountered in developing the SADCA style multilingual support for distributed computing environments, it is important to determine if an approach will work well for both synchronous communications (e.g. Ada tasking) and for asynchronous communications (e.g. PVM interprocess communication). Thus, as a part of the Phase I SBIR research, OTI evaluated alternative techniques with respect to their ability to support both Ada and PVM.

4.6 Comparison of Interprocess Communication Paradigms

Prior to the consideration of techniques to support software testing in a distributed environment, it is necessary to carefully consider and document the characteristic attributes of the possible communications mechanisms. Examination of the range of machine architectures and languages reveals the first consideration should be according to the basic mechanism used for sharing information among processes. From this perspective, the information sharing approaches can be categorized according to shared memory versus message passing.

Many multiple CPU machines have been constructed to support the shared memory paradigm. However, the importance of shared memory is diminished in the distributed computing environment as the technique is generally applicable only within a single multiple CPU machine having its own local shared memory. In addition, although a few modern computer architectures demonstrate a degree of success with shared memory, it has been demonstrated that the use of shared memory is not sufficiently scalable.

Recent trends in multiple CPU computer architecture and network based computing have made significant progress using the message passing paradigm. In this style of computing, communication between processes occurs when one process explicitly sends a message to another. Within the message passing realm, it is also necessary to focus on the distinctions between the synchronous message passing mode of operations versus the asynchronous message passing mode. Further elaboration on these two modes of operation are provided in the following subsections. Understanding of the differences between these modes of communications is essential for subsequent consideration of implementation strategies for ETADS static and dynamic analysis services.

4.6.1 Synchronous Message Passing

In the synchronous message passing mode of operation, explicit coordination of communications activities is performed between the sender and receiver. When the sending process begins the send operation, its execution is suspended while the message is being passed to the receiving process. Its execution resumes after the receiver has acknowledge receipt of the message. Similarly, the execution of a receiving process will be suspended when it executes the receive operation until a message is detected, after which the receiving process will send an acknowledgment message back to the sender and resume execution.

The rendezvous mechanisms incorporated into the Ada language is a prime example of the synchronous communications paradigm. The figure below illustrates a single task, task A, performing a rendezvous with another task, task B. In the Ada scenario, there is a language construct that provides a mutual exclusion zone in the execution timeline. This is accomplished by delaying the receiver's acknowledgment message until the end of execution of the accept code block. Within this region, between the receiver's receipt of the message, and the receiver's dispatch of the acknowledgment message, the execution of sender and receiver tasks is said to be synchronized.

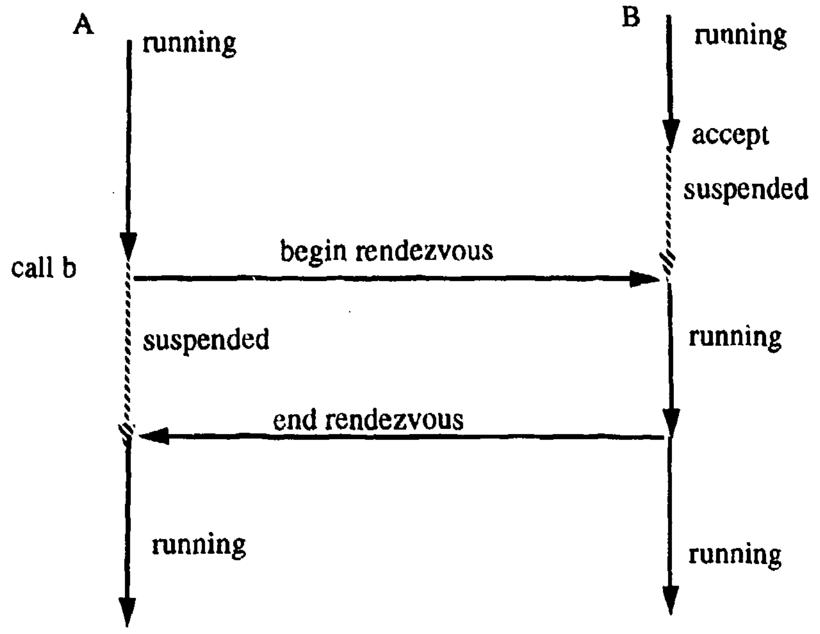


Figure 4.6.1-1 Synchronous Communications with Ada Rendezvous

4.6.2 Asynchronous Message Passing

In the asynchronous message passing mode of operation, no explicit coordination of communications activities is performed between the sender and receiver. When the sending process begins the send operation, its execution is delayed only by the amount of time required for the message to be formatted and inserted into a buffer used by the underlying layer of communications software. Execution of the sending process immediately resumes although there may be a significant delay before the receiving process actually gets the message. From the perspective of the receiving process, there is no direct correlation between the time at which a given message was dispatched versus the time at which the message is received.

Examples of the asynchronous message passing paradigm can be illustrated using the PVM send and receive operations. The PVM receive can be further refined to include two different types of receivers: 1) blocking, and 2) non-blocking. In the blocking example, the call to pvm_recv will cause the execution of the receiver process to be suspended indefinitely until the requested message is delivered. In contrast, in the non-blocking example, the call to pvm_nrecv will cause an immediate return from the subroutine call in either case with a status code to indicate if a message was actually ready.

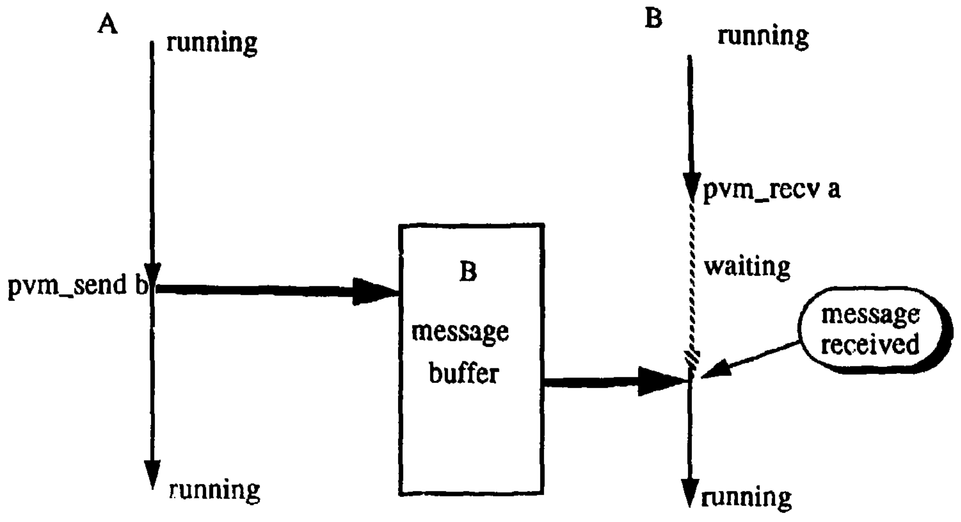


Figure 4.6.2-1 Asynchronous Communications Example: Blocking Receive

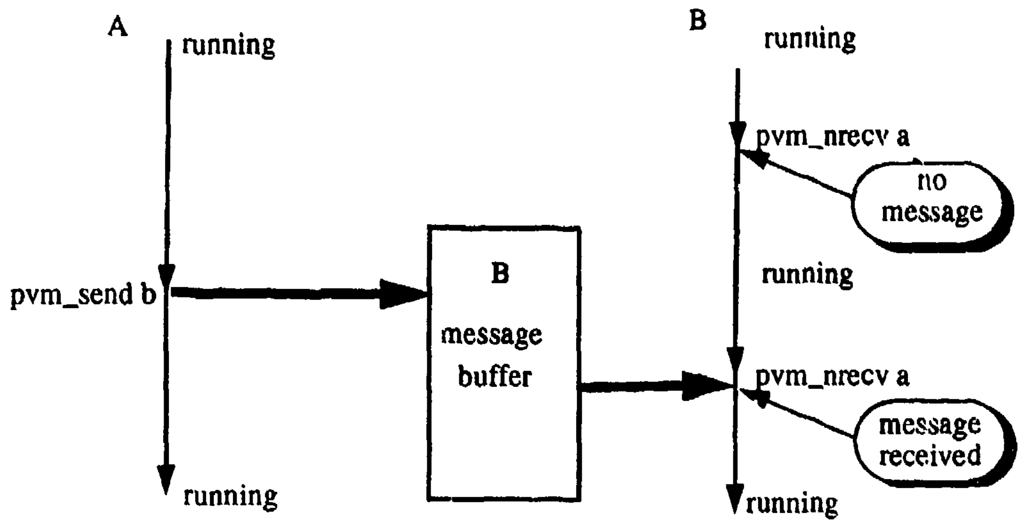


Figure 4.6.2-2 Asynchronous Communications Example: Non-Blocking Receive

4.7 General Purpose Message-Passing Programming Support

The ETADS architecture must be designed to address a wide range of languages and message passing mechanisms including both synchronous and asynchronous communications paradigms. In addition, ETADS must be configured for multiple target languages, which is an objective that is entirely compatible with the successful strategy implemented in OTI's SADCA, to effectively handle remotely communicating processes that happened to be coded in different languages. ETADS will provide the basis for tracing and analyzing the behavior of such multi-language distributed software.

For distributed systems, the general purpose paradigm for communications will usually be based on asynchronous communications mechanisms. Implementation of such communications in current distributed systems usually relies on the utilization of custom communications libraries. In such libraries, a variety of communications services are provided including subroutines to send messages, receive messages, and examine various types of status information. As a prime example of such library utilization, one can point to the interface to the socket I/O library to facilitate network communications in an ordinary network of Sun workstations. The programmer wanting to implement network communications over a Sun LAN/WAN can make subroutine calls to the services in the socket I/O library. Collectively, these services provide the communications backbone for distributed processing.

In general, ETADS will be designed to recognized references to communications services drawn from a communications library. Implementation of the recognition mechanism is dependent on the configuration of the ETADS tool with the user's communications library. Such behavior is readily achievable through a preprocessing procedure similar to what is currently used within SADCA for preprocessing Ada libraries.

In comparison to the use of custom libraries as described above, PVM/C and PVM/Fortran provides a collection of predefined communications services that have been defined to provide the user of the services with a consistent abstract view of a virtual machine. In considering the details of PVM versus other communications libraries, there are clear similarities that have an impact on the ETADS design. The most important similarity upon which the general ETADS design for distributed software is drawn is the recognition that the PVM facilities themselves are organized precisely as a library of communications services. Thus, the major structural characteristics exhibited by PVM are the same as for common off-the-shelf communications libraries. The differences can be considered to consist of relatively minor variations in specific communications services offered by a particular library. As has been demonstrated with OTI's multilingual approach incorporated in SADCA, one can expect the possible set of unique operators to form a relatively small set of additional primitives as extensions to the current set of language primitives for the sequential design. For example, support of the distributed software environment introduces a variety of communications methods such as those provided by PVM including: Point-to-Point, Multicast, and Broadcast send/receive operations. It can be anticipated that there is a very limited number of possibilities for additional unique communications primitives. This will be true even with the incorporation of emerging technology such as that contained in the MPI.

5.0 Phase I Results

The Phase I research effort was approached with the goal of developing and evaluating the feasibility of innovative new techniques for testing distributed real-time software. The overall objective is to extend to this new domain techniques comparable to those implemented in SADCA for the sequential software domain. The distributed real-time software introduces a variety of complex new technical issues that are not encountered in the sequential domain and thus are not addressed by the existing SADCA/ METAsoft tools. However, the SADCA technology baseline can clearly form the nucleus for a new generation of multilingual software analysis tool providing integrated support for testing distributed software. While pursuing this objective, OTI's research effort was organized along the major functional areas that had been identified earlier as potentially high payoff candidates for Phase I research. In addition, further analysis of the distributed software testing domain during Phase I identified a set of interrelated supporting technology areas, each of which demands special attention. The tables below list the functional areas researched during this effort and presents the related supporting technology areas. Each of these areas will be described in further detail in the listed subsections.

Table 5.0-1 Major Technical Areas

CHAPTER	TITLE
5.1	Static Concurrency and Data Flow Analysis
5.2	Timing Analysis
5.3	Deterministic Execution Testing
5.4	Dependency Analysis
5.5	Mutation Analysis
5.6	Control Flow Testing
5.7	Network Communications Testing
5.8	Problem Tracking and Reporting

Table 5.0-2 Related Technology Concerns

CHAPTER	TITLE
5.9	Distributed Clock Synchronization/Compensation
5.10	Time Stamp Generation
5.11	Minimization of Probe Effects
5.12	Trace Data Compression
5.13	Trace File Management

In general, the technical approach applied under the Phase I ETADS development can be described in terms of the following set of analysis steps applied to each of the technology areas as described in the Phase I proposal. For each technology area identified for research (i.e. those candidate technologies described in the Phase I proposal):

- Define data requirements for desired analysis features and reports
- Define data collection requirements
- Define and analyze steps required to implement data collection
- Define alternatives for implementing data collection steps and assign priorities based on estimated effort for implementation
- Enumerate existing SADCA data collection capabilities
- Comparison of ETADS data collection requirements vs. SADCA capabilities

The general approach applied by OTI to establish confidence in the technical feasibility of the proposed approach addresses the known limitations of predicting the performance of new and unproven technologies. OTI's evaluation of candidate ETADS technologies is based on the identification of alternative evaluation strategies and employing the most appropriate evaluation technique for the given situation. General alternatives that might be applied in individual cases include:

- System modeling,
- Analytic proof techniques,
- System performance simulation,
- Parameterized system performance estimation.

Application system modeling has been used extensively during the Phase I effort. In general, this technique facilitates increased confidence in the completeness and consistency of proposed system concepts through the construction and analysis of model(s). During ETADS modeling and system definition, OTI has used the detailed design of the existing SADCA tool as the baseline for system model construction. At each step during this process, required ETADS definitions and processes were compared with existing SADCA processes to check the feasibility of proposed new functionality. In addition, knowledge of performance trades gained during SADCA development was used in estimating impacts of proposed approaches to specific ETADS design decisions.

In general, the application of analytic proof techniques has not been deemed appropriate for a large system such as ETADS. Although the state of the practice in formal proofs does not easily lend itself to such large complex systems, these techniques may be applied on a limited basis for specific subproblems within the bounds of the complete system.

The application of system performance simulation for the Phase I research effort was an option that was not pursued. In principle, there are some important performance parameters that could be gathered using this technique. For example, memory, CPU, and secondary storage resource utilization due to trace data collection could be evaluated. However, there are also severe limitations on the ability of this technique to generate data that will prove useful in the general case. For example, without going to the expense of simulating performance, our experience with SADCA technology and informal extrapolation to the ETADS environment indicates a very strong relationship between the size of the system under test and the size of the trace files. Furthermore, OTI experience also indicates potentially large variation in resource utilization as a function of the efficiency of the trace data collection techniques employed. Further variation in performance and resource consumption is also predicted as an explicit feature of ETADS under the direct control of the user. Thus, the general utility of the simulation technique for evaluating feasibility and performance characteristics of ETADS is diminished to a certain degree. As a result, the importance of this technique was deemphasized during the Phase I effort.

The application of parameterized system performance estimation is an analytic techique employed as an alternative to or in addition to system simulation. In this approach, a mathematical model of each performance characteristic to be evaluated is constructed in the form of one or more interrelated mathematical formulas describing the estimated performance. In constructing each formula, the implicit/explicit assumptions are described and how they are incorporated into the model is documented. Specific parameters in the model are identified and described as independent variables (e.g. system inputs) in determining the performance. Given the mathematical model of system behavior with respect to one or more specific system parameters, the model is evaluated for a range of values of those variables and the results are examined to provide insight into the anticipated system performance. Areas where this technique might be applied to the ETADS research include evaluation of: variations in ETADS data repository size versus the size of the system under test, variations in timeline perturbation as a function of software probe executions, variations in timing analysis accuracy as a function of distributed clock accuracy.

The system model construction and model analysis approach is the principle technique applied under the Phase I research effort. This is principally due to the combination of the magnitude and range of the proposed extensions to existing SADCA technology and the limited amount of effort available for Phase I research.

The remainder of Section 5 presents the results organized according to the technology areas as listed in Table 5.0-1 and Table 5.0-2 above. This organization allows the presentation of research results along the lines of major functionality that is directly visible to a potential user of the ETADS tool and discussion of the supporting technology issues is reserved until the latter subsections.

5.1 Static Concurrency and Data Flow Analysis

The purpose of static concurrency testing and data flow analysis is to detect faults arising from the misuse of concurrency constructs resulting in run-time anomalies such as deadlock or infinite waits. Several alternative methods for static analysis of software composed of a set of processes and tasks have been reviewed and the ability to generalize the techniques for implementation within ETADS has been evaluated.

5.1.1 Objectives and Benefits

Static concurrency analysis is a computer automatable analytical process that focuses on identifying and reporting the characteristics of the structure of concurrent processes. Although there are certain conceptual similarities between this form of analysis and traditional static analysis for sequential software, static concurrency analysis focuses attention on the relationships between concurrency states of interrelated processes. The main objective of the analysis process is the identification and reporting of concurrency states that correspond to an error condition or may subsequently lead to an error condition. Potential errors of interest in this area include deadlock conditions and various forms of race conditions. In general, deadlock conditions occur when one or more processes enter a wait state in which they are waiting on the occurrence of an event that cannot occur. In comparison, concurrency analysis may also be applied to the identification of situations in which multiple processes may modify a data item without any intervening reference to the data item (i.e. a race condition). Such situations usually indicate the existence of an error since the results of some computation are effectively discarded or overwritten. The static concurrency analysis can be an effective tool for identifying software defects prior to the execution of test cases on the SUT. The ability to detect potential concurrency problems early can be quite valuable for large distributed software as it has the potential to reduce the number of software defects that are encountered late in the life-cycle.

Data flow analysis is another computer automatable analytical process that focuses on identifying and reporting the structural characteristics concurrent processes. In contrast to concurrency analysis, this technique focuses on analyzing the flow of data between different processes. Traditional data flow analysis techniques, like those implemented in OTI's SADCA, are intended to identify data usage anomalies within a single sequential program. The data flow analysis techniques researched under this SBIR effort are intended to extend the data flow analysis capabilities to include interprocess data flow, such that anomalies in data usage across process boundaries can be identified and reported. As with concurrency analysis, the potential benefit from data flow analysis is large as it can dramatically reduce the amount of effort required in software testing. Static concurrency and data flow analysis are presented together because they both rely on similar data gathered from the analysis of the static properties of the SUT (i.e. the are related forms of static analysis).

5.1.2 Technical Issues

The development of specific techniques for ETADS to facilitate the desired automated analysis features requires the detailed consideration of each of the following technical issues:

- Synchronous vs. Asynchronous communications
- Abstract graph representation/modelling
- Enumeration vs. analytic representation

- State space reduction
- Shared memory vs. distributed memory
- Anonymous process interactions

As mentioned earlier, there are major operational differences between synchronous communications versus asynchronous communications. Further consideration of these differences reveals a profound impact on the feasibility of certain analysis techniques.

Alternative techniques exist or can be derived to serve the role of abstract graph representation/ modelling. Alternatives investigated in this research include Taylor's enumerative approach and a more general Petri net modelling approach.

Two general categories of approaches for state space representation in support of concurrency analysis include: 1) enumeration - in which all of the possible combinations of states are enumerated, and 2) analytic - in which some form of symbolic representation is derived in place of listing/ enumerating all possible combinations.

In general, existing approaches lead to very large state spaces to be analyzed, leading to a strong need to apply state space reduction techniques wherever possible. This is a major issue in all of the current approaches.

Consideration of techniques for both concurrency analysis and data flow analysis are heavily impacted by the inclusion of shared memory as opposed to distributed memory. Shared memory environments exhibit much stronger coupling between processes as each shared data item directly contributes to the size of the program graph and concurrency state space.

Another issue that must be addressed in considering alternative techniques is the degree to which communicating processes are statically and/or dynamically identifiable. Instances where processes cannot be statically identified must be described as anonymous process interactions and the analysis techniques must treat such process interactions specially.

5.1.3 Research Results

The technical approach for Phase I research can be described in terms of the following set of analysis steps as applied to a variety of alternative algorithms:

- Define data requirements for desired analysis features and reports
- Define data collection requirements
- Enumerate existing SADCA data collection capabilities
- Comparison of ETADS data collection requirements vs. SADCA capabilities

Schedule and budget considerations mainly limited the selection of feasibility evaluation techniques to manual analysis with minor consideration for algorithm design. Algorithm design was limited to the identification of required steps to integrate alternative techniques into the SADCA tool architecture.

Several alternative strategies were identified and evaluated with respect to ETADS integration. These strategies are further described in the following sections and include:

- Taylor's algorithm,

- Petri net modelling,
- Extended Concurrency Graphs (ECG),
- Shared-Variable Concurrency Graph (SVCG),
- and others.

Analysis of integration potential with SADCA structures reveals that Taylor's algorithm can be readily integrated SADCA based mainly on the existing structures. However, it is also clear that Taylor's algorithm is only applicable to synchronously communicating languages such as Ada. In the more general case of distributed systems in which the communications may be predominately asynchronous, the study results indicate that Petri net modelling and analysis techniques are more appropriate.

5.1.3.1 Analyzing Synchronously Communicating Processes

5.1.3.1.1 Taylor's Algorithm

Detailed analysis of the requirements for Taylor's algorithm for synchronization sequence analysis reveals a feasible algorithm that can be readily integrated with existing SADCA technology with suitable extensions to the current SADCA architecture. However, its application must be limited to the analysis of identifiable processes that communicate only via synchronous communications primitives. The algorithm cannot be applied in environments where some of the communications are performed in an asynchronous manner. Unfortunately, the predominate communications technology for distributed systems is asynchronous, and therefore Taylor's algorithm is seen as having relatively limited value in the general purpose distributed domain.

Taylor's algorithm requires as input, task graphs with state nodes indicating tasking activities. Included in the set of possible task related states are entry calls, accepts, selects, delays, aborts and task declaration activities. A complete program is represented as a collection of task graphs. For comparison, these input requirements are comparable to the current structure diagrams constructed and maintained by the existing SADCA tool.

Taylor's algorithm constructs a concurrency state graph in which each node is a tuple representing the current state of every task in the system. Edges represent transitions from one concurrency state node to the next. Analysis of paths through this graph detects reachable states that represent deadlock or infinite wait. Sequential sections in each task that occur between transitions in the concurrency state graph represent activities such as variable accesses that may be performed in parallel with other tasks. The nodes of the task graph represent concurrency constructs within the task it models. It is critical to realize that this algorithm is intended for languages using a rendezvous mechanism or other synchronous communication facility. Also, the destinations of communications or entry calls must be statically determined. For Ada code, this is straightforward in that the task which is the object of an entry call is often statically identifiable. Sequential components of a task are not important at this point.

The concurrency state graph is built by enumerating all valid combinations of states in which tasks could exist. As a result, Taylor's algorithm is applicable to systems using the rendezvous protocol or synchronous communication where the destination is named. The requirement that the communication be synchronous arises from the assumption that once a task enters a state it remains there until the concurrency activity associated with it completes. For example, a task requesting an entry waits until the rendezvous is completed before proceeding. All feasible combinations of these task states must be examined to report faults related to concurrency.

Although the algorithm eliminates as many impossible states as it can, the state space size can still grow exponentially. As a result, current research efforts have focused on minimization of the size of this graph. A promising method under investigation for dealing with the graph size is to parcel a large system of processes into smaller subsystems and analyze each of them separately.

Detailed analysis of the requirements and design for Taylor's algorithm indicates the technique is automatable within the ETADS framework for synchronously communicating systems of moderate size, where size is measured in terms of the number of task activity nodes in the concurrency graph (i.e. not to be confused with size in terms of lines of code). To facilitate its implementation for more complex systems (those with higher levels of concurrent activity) efforts must be directed at pursuing techniques for reducing the size of the concurrency state space graph as well as parceling a system into smaller systems for individual analysis. One candidate technique for reducing the state graph's size is that of Long and Clarke which has also been considered for integration into ETADS.

5.1.3.1.2 Long/Clarke Modifications to Taylor's Algorithm

The Long and Clarke adaptation of Taylor's algorithm (LCAT) is designed to eliminate unnecessary states from the original task graphs, thereby significantly reducing the number of combinations of concurrent states.

Detailed examination of Taylor's algorithm reveals that the definition of what constitutes a "state node" can be considered too restrictive. If has the beneficial effect of simplifying the construction of the concurrency graph, however, it does so at the expense of creating more states than is absolutely necessary. Furthermore, in real-world applications of concurrency analysis the size of the graph proves to be of paramount importance in determining if the automated analysis function will complete within a reasonable time delay and with a reasonable level of resource consumption.

As a consequence of the definition of "state node" in Taylor's algorithm, a program containing only a few statements that actually perform any form of interprocess communications may result in a very large number of "concurrency states". For example, a subroutine call statement will be classified as a state node if any subroutine in the transitive closure of its called subroutines performs any actual concurrency operations. In generating the set of concurrency states, each of the state nodes on the execution path (i.e. including all of the state nodes derived from subroutine call statements) to the state node that corresponds to an actual tasking activity (e.g. an Ada task entry call) will be used in be used in generating the set of all possible combinations of states across all of the tasks. The ramifications of the combinatorics is clear when one considers the example of a simple program involving only two tasks and only one instance in the code where task entry call is made, but the task entry call is made after a sequence of several subroutine calls within the calling task and several subroutine calls within the accepting task. Direct application of Taylor's algorithm results in a concurrency state space in which the various combinations of intermediate subroutine calls considered to be important, identifiable states to be reported in addition to the concurrency state in which an actual task interaction occurs. The focus of the algorithm modification in LCAT is the elimination of the intermediate states for which we have less interest.

LCAT eliminates a potentially large number of intermediate concurrency states by redefining the meaning of a node in the graph to construct a "Task Interaction Concurrency Graph (TICG)".In such a graph, the meaning of a node is defined to correspond to program execution within a region of code between two task interaction statements. For example, a single region may correspond to the complete set of paths between two task entry calls. In contrast to Taylor's algorithm, the existence of subroutine calls and/or scope block entry/exit on a path is of no concern in identifying concurrency states. All of such points that Taylor's algorithm would treat as distinct states are collapsed together into a single state.

The immediate benefits associated with LCAT implementation over Taylor's algorithm is the drastic size reduction of the graph. However, the expense of this approach is in incurred in two areas: 1) the algorithm to identify regions from which nodes are constructed is more complex, and 2) the correlation of concurrency states with the user's original software is less intuitive for the user. The first concern is relatively unimportant as the complexity is manageable and the node construction process is subsequently automated by the computer. The second concern is of greater significance since it impacts the human-machine interface. If the technique is to be applied to maximum advantage, additional graphical user interface mechanisms must also be developed to facilitate the human understanding of the correlation of concurrency state nodes and the original software.

The remaining concern with LCAT with respect to application in the distributed software environment is it too is restricted to analysis of software in which communications are exclusively synchronous. Distributed systems are often restricted to asynchronous communications mechanisms, which precludes the use of either Taylor's algorithm or Long/Clarke's modification. On the other hand, it may be of significant value to a software engineer to use such techniques for analysis of a subsystem within which communications are performed synchronously. Although such analysis would necessarily be incomplete (i.e. since asynchronous communications beyond the boundaries of the subsystem would be ignored), the results could be beneficial in identifying errors and/or unexpected behavior.

5.1.3.2 Analyzing Asynchronously Communicating Processes

The extension of concurrency analysis techniques to include the asynchronous communications paradigm is extremely important for the distributed software environment since this is the predominant form of communications. However, Taylor's method and related approaches such as LCAT are not applicable for PVM-like environments for two important reasons: statically unidentifiable message destinations and asynchronous communication. In languages such as PVM it is often difficult or impossible to statically detect which process is the destination of a send operation. For example, consider the partial function prototype for the send operation in PVM given below:

```
pvm_send (int tid,...)
```

This send routine causes the message in the current buffer to be sent to the process indicated by the input integer parameter **tid**. **Tid**'s are task identifiers assigned by PVM at run time and the user has no control over these values. Although there are query routines that are available at run time to determine what tids are currently active, this does not solve the static problem. Because the **tid** is simply an integer variable and is not limited to any statically detectable values, it is extremely difficult to determine which process(es) is (are) the destination. Therefore, Taylor's algorithm would have to assume several processes as potential destinations and perform analysis on all cases. Not only does this tremendously increase the number of possible task states, but analysis may derive results based on communications that would never actually occur. The characteristics of this example are important to note because many distributed communication support libraries have the same characteristics as the PVM send routine.

The second obstacle to applying Taylor's algorithm in an asynchronous communication environment like PVM is the fact that message send operations are asynchronous. As noted in the preceding paragraphs, the method relies heavily on the fact that communication must be synchronous. Since this technique is not applicable to many distributed environments, OTI has identified a representation more suitable for analyzing systems containing asynchronously communicating processes which is based on Petri Nets. Petri Nets have been widely used to model Ada-like systems and are more generally applicable to most kinds of distributed systems, whether synchronous or asynchronous in communication.

Petri Nets are graphs composed of two types of nodes: places and transitions. A place in a net would corresponds to a single state in which a process may exist. A transition represents the change from one state to another. One can represent a process as a set of interconnected places and transitions; combining all process nets results in a single Petri Net describing the potential activity of the entire system. Since Petri Nets are a well-defined modelling technique, there exist algorithms to compute the reachable states of a set of nets for a given system. A reachability state graph can be generated by examining all possible transitions that may occur from a particular state. The reachable states can be examined for situations such as deadlock.

Although the reachability state analysis of petri nets can also produce large numbers of states, OTI has identified a promising approach to analyzing Petri Nets for large systems. By organizing a system of Petri Nets hierarchically where each level in the hierarchy is analyzed separately, the number of combinations of states that must be considered at any given time is greatly reduced.

5.1.3.3 Data Flow Analysis

Data flow testing has been shown to be an effective mechanism for detecting certain variable usage errors. Although variable usage errors (e.g., variables used before initialized, variables never used, etc.) occur in sequential programs, these problems are considerably more complicated for parallel programs, where different execution threads reference shared variables. The exact order in which processes refer to a shared variable can influence the results produced by the program. Various approaches to detecting race conditions with respect to shared variables are described below. A race condition for a variable exists when two or more processes may access the variable in a non-deterministic order and at least one of those accesses updates the value.

5.1.3.3.1 Graph Based Approaches

Graph-based approaches are most suitable for incorporation in a general purpose tool since they typically do not require symbolic execution, which is generally cost prohibitive. The extended concurrency graph (ECG) models a complete SPMD (Single Program / Multiple Data) program as one task. SPMD programs are usually characterized by the use of fork/join constructs, where multiple copies of the same code are caused to execute concurrently. The ECG is a task graph including all references to shared variables. By creating duplicates of this task and considering all valid pairs of task states, one can analyze all of the potential concurrency states that could exist. Because all tasks are essentially the same, only two copies (tasks) are necessary to perform analysis. Variables that can have a data race are easily detected from the combined graph. However, because this graph could grow quite large, a simplified version has been developed, the shared variable concurrency graph (SVCG). The SVCG is the same as the ECG except it contains only nodes related to a single variable. By analyzing each SVCG one at a time, the size of the graph being analyzed is considerably smaller.

When this approach is extended for general distributed programs where each task may be different from the others, the end result is the same as the concurrency state graph generated by Taylor's algorithm for static concurrency testing. Concurrency states can be examined to determine if multiple tasks could access a particular shared variable.

5.1.3.4 Intra-Process Data Flow Analysis

Because data flow analysis utilizes essentially the same graph-based techniques as static concurrency analysis, investigation into optimization techniques intended to solve the static analysis problems will also focus on their applicability to the data flow analysis problem. One should note that it is also quite feasible to perform traditional data flow analysis at the process level. Since a single process is essentially a sequential program, these data flow techniques can be used to provide useful information about the behavior of processes within the system. Several sequential data flow analysis techniques have been implemented in the current SADCA test tool that are applicable at the process level as well. Also, since in most cases distributed systems do not characteristically use shared memory, sophisticated interprocess data flow analysis may often be unnecessary.

5.2 Timing Analysis

The analysis of timing constraints is one of the more difficult tasks when testing real-time software systems. OTI has researched several techniques that specifically address the statistical nature of software performance with respect to the satisfaction of real-time constraints. These techniques provide information directly related to real-time constraints imposed on the software under test.

5.2.1 Objectives and Benefits

The objective of timing analysis is to focus attention on the timing characteristics exhibited by the software at run-time. These characteristics can be very important factors in the correct behavior of real-time software, distributed software and real-time distributed software. By definition, the timing relationships of real-time software are in some way explicitly included in the specification of the correct behavior. That is, the real-time function being computed is in some way a function of time in addition to the other explicit parameters: $F=f(x, y, z, t)$. For distributed software, although the performance of the system may not be defined with explicit timing relationships, the available implementation technology involving asynchronous communications typically introduces potential variability in timing due to race conditions. For real-time distributed software, these effects are combined to further complicate matters. In any case, it is important for the developer to have the ability to observe and evaluate these timing characteristics. Furthermore, it is particularly important for real-time system developers to be able to do so and compare the timing observations with respect to the real-time constraints.

Whether or not it is automated, timing analysis is an important part of the development of real-time systems as well as distributed systems. The correct operation of the system is dependent on time to some degree. Furthermore, fielded systems typically operate under a range of real-world scenarios for which a wide range of behavior may be observed. Timing analysis is important to fully evaluate the performance across the entire spectrum of environmental conditions in order to properly evaluate the system reliability.

5.2.2 Technical Issues

Consideration of alternative techniques to provide the desired timing analysis services must deal with a range of important technique issues including those listed below

- Clock access
- Clock resolution
- Clock drift
- Communications delay
- Resource utilization due to time-stamp recording
    - CPU
    - Primary memory
    - Secondary storage
    - Network I/O

In a distributed system environment involving heterogeneous computing resources, uniform access to a clock may or may not be available at every processor. For example, some systems may have a very precise clock, some systems may have a crude clock, and other systems may have no clock at all. ETADS is designed to function within the limits imposed by the system under test.

Given access to clocks, ETADS must also address the variations in clock resolution and clock drift. Once again, there may be large differences in clock resolution.

In addition to clock resolution, variations in clock drifts may play an important role in the post-experiment resolution of time stamped events.

In defining ETADS procedures for correcting time stamp errors, large variations in communications delays should be considered in clock compensation mechanisms.

Resource utilization due to time-stamp recording must be carefully considered in all design decisions of ETADS development. This issue is particularly important with respect to the instrumentation concepts to be implemented as it can have large effects on resource utilization. CPU resources are consumed with each call to a clock function. Memory is consumed due to the buffering of time-stamped event data. Secondary storage space is consumed with the recording of time-stamped event data in trace files. Network communications bandwidth is consumed during the process of transferring trace data back to the central ETADS host machine. Minimization of resource utilization must be given high priority during design.

5.2.3 Research Results

The timing analysis techniques developed for ETADS are designed to provide the user with a variety of visualization mechanisms intended to enhance identification of timing anomalies. Some visualization tools are designed to provide a higher level abstract view of the software performance, after which other visualization tools can be employed to focus on a more narrow portion of the SUT.

The approach in developing techniques supporting timing analysis relies on system modeling and analysis. Initial concern is the definition of the useful services that are directly visible to the ETADS user (i.e. definition of data analysis requirements and reporting requirements), after which the sequence of supporting analysis activities are performed. These activities include: definition of data collection requirements, definition of requirements for automated data collection, and comparison of ETADS data collection requirements versus existing SADCA data collection facilities.

Evaluation of the feasibility of each ETADS capability is supported by the direct comparison with existing SADCA services. Such comparison facilitates the identification of expected resource utilization based on similar SADCA functionality and serves to facilitate the estimation of effort required to implement the new functionality as an integrated SADCA service.

5.2.3.1 Timing Reports

Potential timing analysis reports include: normalized process execution distribution report, process execution times report, system performance summary report, and other process event measurements.

5.2.3.1.1 Normalized Process Execution Distribution Report

A normalized process execution distribution diagram shows each process' execution performance relative to its normalized timeline. It highlights any processes that violate user-supplied elapsed time constraints. Generation of this diagram requires minimal probe insertion and local process time values only. Figure 5.2.3.1.1-1 presents an example graphical report of this type.

Data Requirements

- Trace data showing begin/end time for each run of the task
- Task time limitation constraint, supplied by user

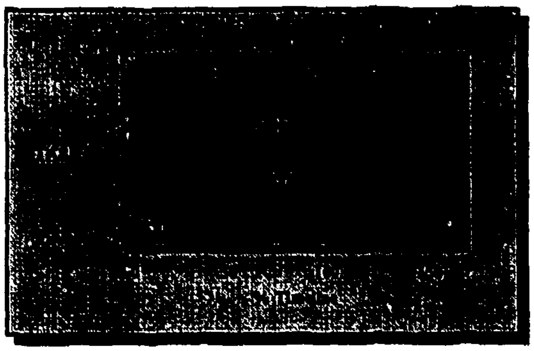


Figure 5.2.3.1.1-1 Normalized Process Execution Distribution

Collection of this type of timing data does not require that different processor clocks be related to each other. All that is required is the knowledge of how to query a task's clock and the unit of time it measures if necessary. Knowing syntactic details of how to query clocks is necessary to generate any of the proposed timing analysis reports. Therefore, even if the language translators are independent of system details, inserting timing collection instruments will be somewhat system dependent and should therefore be a template driven process. ETADS must provide an interface that allows the specification of the format for querying a clock on each processor used by the system under test. The translators can then use these specification templates when generating timing measurement instrumentation.

5.2.3.1.2 Process Execution Times Report

The process execution times report presents similar information, but rather than displaying normalized time scales, all data is presented relative to the overall system timeline. Again, minimal timing instruments are required. However, local process times must be resolved to a global measure of elapsed time. Figure 5.2.3.1.2-1 presents an example graphical report of this type.

Data Requirements

- Trace data showing begin/end time for each run of the task
- Task time limitation constraint (supplied by user)
- Ability to construct a global ordering of time-stamped events

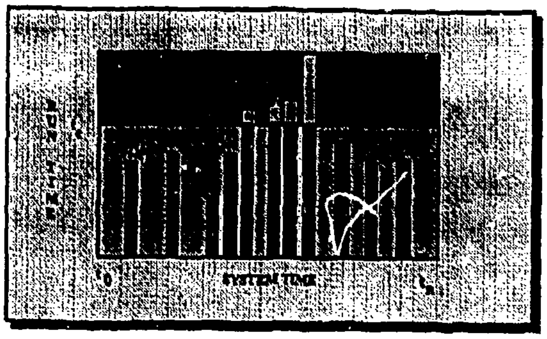


Figure 5.2.3.1.2-1 Process Execution Times Report

5.2.3.1.3 System Performance Summary Report

The System Performance Summary Report constructs an aggregate view of the performance of many tasks in relation to the system time line. This is an alternative view of the same performance data gathered for other timing analysis reports described above, thus facilitating a global view of task performance. Figure 5.2.3.1.3-1 presents an example graphical report of this type.

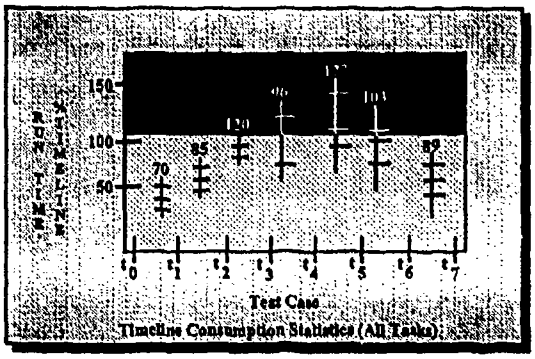


Figure 5.2.3.1.3-1 System Performance Summary Report

Data Requirements

- Trace data showing begin/end time for each run of the task
- Task time limitation constraint (supplied by user)
- Must be able to construct a global ordering of time-stamped events

5.2.3.1.4 Process Event Measurement Report

In addition to these specific large grained timing reports, OTI has identified several fine grained approaches. These reports include interprocess communication delays and interprocess event time lines, as well as reports detailing how much time a process spends waiting for communications or rendezvous. Because finer levels of detail such as these examples generally require more timing probes, the user must be provided with control over the level of measurement desired.

5.2.3.1.5 Alternative Timing Analysis Reports

In order to support graphical replay of events and deterministic execution, ETADS will have the ability to trace the occurrence of events. By adding a time-stamp to the trace data for events, ETADS would have the ability to generate more fine-grained timing reports. These may include:

- inter-task synchronization delays, e.g. - the time difference between the request for synchronization and its completion such as the BEGIN and END of a rendezvous (see Figure 5.2.3.1.5-1)
- delays between intra-task events (i.e. delays between task events within a single process, not related to system timeline)
- begin/end of a particular event such as a wait to receive a message
- percentage of time a task is in wait states (see Figure 5.2.3.1.5-2)
- simple event reports showing the system times at which specific events occur. This illustrates each instance of a particular event class

Note that as in SADCA, not only can the tabular event reports be generated for a whole program or process, but the user could also select an event of interest in a task graph to pull up that event's timing data.

Figure 5.2.3.1.5-1presents another alternative timing analysis graph that facilitates a detailed view of event timing relationships. The focus of this example is the illustration of inter-task synchronization event delay (e.g. Ada task rendezvous). The arrows indicate synchronization relationships, points represent the events of interest mapped onto the system time line. Without timing data, the related events can still be indicated with arrows to show the relative ordering, but the relative magnitude of the delay between them cannot be evaluated. A scrolling window can be used to scan along a long range of data. The dashed portions of the lines in the figure indicate when the process is inactive. In such a figure, each instance of a process or task type can be shown separately or grouped together into one representative process. In that case, process creations could be illustrated as events in the process being created, and dashed lines can be applied to indicate when a process does not exist. This may be a useful feature in addition to showing each instance of the process separately, so that the user can choose the representation best suited to current needs.

Figure 5.2.3.5-2 presents another timing analysis view that provides insight into process performance with respect to process states (i.e. existence, active, waiting). The diagram allows the user to focus on areas where processes are idling. When seeking to improve overall system performance< for example, idle time can often be reduced through a reorganization of interprocess communications.

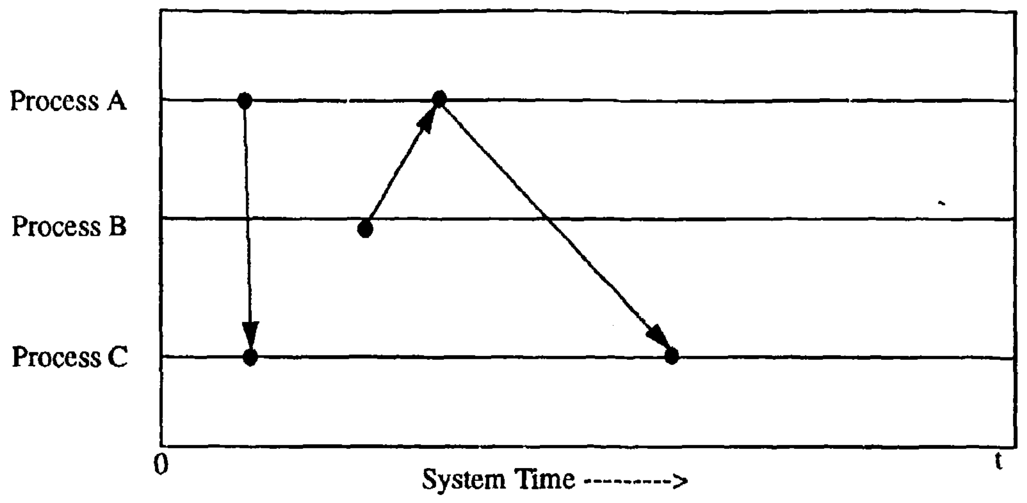
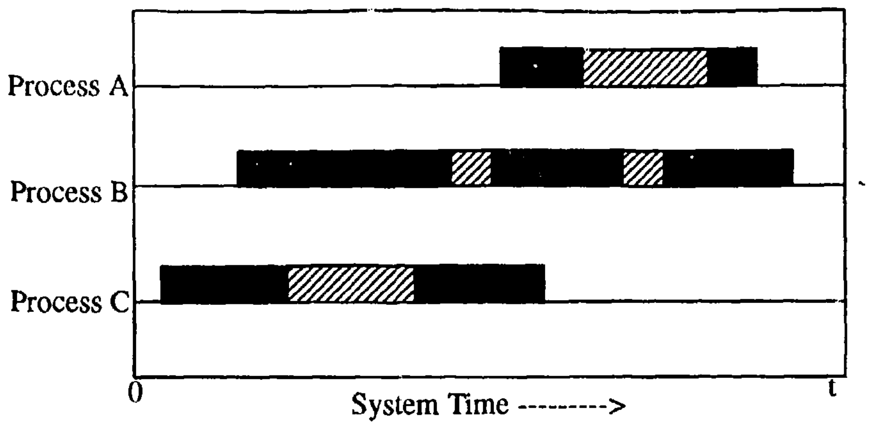


Figure 5.2.3.1.5-1 Inter-Task Synchronization Event Delay

This timing analysis view provides the means for identifying sources of delay on the overall system time line. In addition, other process states could potentially be defined such as: busy waiting, interrupted, or event handling, based on the recognition of specific types of events from the event trace.

It is also notable that these timing analysis graphs are not restricted to display relative to the system time line. Each process' trace data may be presented in the same graphical form, but with the data scaled as a percentage of the process' time line as shown in Figure 5.2.3.1.5-3. In comparison to the previous figure that facilitates the relative evaluation of timing between processes, this form of timing display focus more on performance with respect to individual tasks.

The feasibility of implementing timing diagrams such as these is not of major concern. Similar diagrams have been produced by other tools, usually through the application of hardware monitors or system dependent monitoring mechanisms. In contrast, ETADS is designed to implement similar facilities but in a portable multilingual fashion as has been accomplished with SADCA technology. The ETADS concepts are based on the application of software probes to collect the requisite data. Therefore, we have selected a set of reports that can be generated from data collected by software probes. Also, we have selected timing reports that will be highly valuable to software test personnel who are working with real-time systems having challenging timing constraints. In addition, we stress the ability to gather and present timing data that is pertinent across multiple levels of abstraction from processor level, to process level and finer granularity.

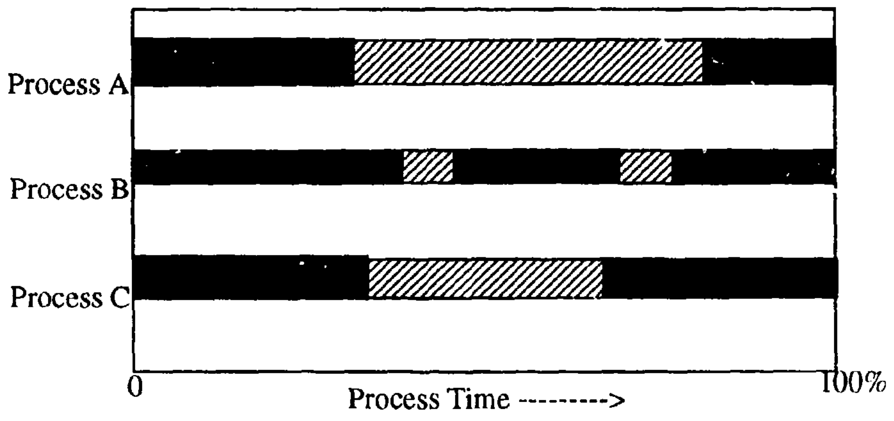


■ Active

▨ Waiting

— Process not alive

Figure 5.2.3.1.5-2 Example Process State Diagram



■ Active

▨ Waiting

Figure 5.2.3.1.5-3 Process State Percentages Diagram

System

5.2.3.2 System Configuration Details

In order to relate process time data to each other, several system details must be collected. These include:

- a description of the system processor connection which may be specified by the user;
- the ability to access clocks to record timing data dynamically;
- the ability to identify process to processor assignments;
- the ability to relate many clocks to a single global timeline.

Each of these issues are addressed in the following subsections:

- 5.2.3.2.1 System Architecture Specification
- 5.2.3.2.2 Process to Processor Assignment
- 5.2.3.3 Resolving Distributed Clocks

5.2.3.2.1 System Architecture Specification

All of the above timing analysis reports require collection of time stamps during program execution. In other words, instruments must be able to query processor clocks. In order to provide a general purpose testing environment, the user must be allowed to specify a system's configuration of processor and communications resources as well as the format of clock measurement functions on each processor. This can be done by allowing the user to create function templates specifying the syntax of clock measurement for each language used by a distributed program. Along with information regarding process-to-processor assignment, this provides sufficient detail to automatically customize timing measurement instruments for each process to be measured.

5.2.3.2.2 Process to Processor Assignment

In systems written in PVM it is possible to trace onto which host a particular task is spawned or created. The *pvm_config* and *pvm_tasks* functions return information containing host PVM tids and task tids for requested tasks. These tids could also be recorded and passed to the trace file. Post processing can then relate each process' trace data to a particular machine.

However, this mode of operation is not always possible as there will be distributed systems where it is not possible to detect dynamically to which processors a given task is assigned. For example, consider a distributed set of programs written in many different languages or a language that does not support explicit process spawning. In this case, ETADS must provide an interface allowing the user to specify the system configuration. In either case, ETADS will need to know a certain amount of basic information concerning the distributed system environment. For example, ETADS must know the processors that exist in the target environment and how they communicate in order to translate a system of programs running on them.

ETADS must implement a sophisticated strategy for tracking process identifiers from the target environment over the life-span of the distributed systems execution such that specific instances of a given type of process can be uniquely identified and reported. For example, PVM based systems use a unique tid value (i.e. Task Identifier) for each processed created via the pvm_spawn subroutine. ETADS must track these tid's through the life span of the distributed program to facilitate reporting. Since the PVM system controls the assignment of tid values upon process creation and tid values may be reused as old processes are terminated and their tid values are returned to the pool of available values, ETADS must maintain its own indexing scheme for tracking and reporting process identifiers. In comparison, similar operations are necessary to maintain a task global index counter to record trace data with respect to a particular instance of a task type. In general, whether PVM, Ada, some other language, or a mixture of languages in a distributed environment, ETADS must maintain its own process indexing scheme that uniquely identifies processes throughout the system. The ability to identify processes uniquely is not only critical to timing analysis but must be available for any correlation of run-time behavior data. As such, solutions implemented for collection of timing analysis will be used for all dynamic analysis functions presented later in this report.

5.2.3.3 Resolving Distributed Clocks

Fundamentally, the accuracy of timing analysis results are related to the accuracy of timing measurements taken throughout the distributed system during execution of the SUT, and therefore special consideration must be given to techniques to maximize the degree to which accurate clock measurements can be performed. However, the consideration of specific techniques to be integrated into ETADS must also be performed with the clear goal of supporting the ETADS user with general purpose tools that handle a wide spectrum of distributed computing environments with the maximum possible degree of automation. Supporting both of these requirements in combination requires that ETADS place strong emphasis on defining and integrating automated support for resolving differences in distributed clocks and providing automated compensation for those differences.

For all but the most trivial timing analysis reports, each process' time stamps must be related to some overall global system time. However, in a distributed system each processor has its own local clock; there does not usually exist one global clock that all processes may access. Therefore, once a test is completed, each process' individual time stamps must be resolved. In order to accomplish this operation two additional items of data are required with respect to the test: 1)which processor each process executes on and 2) a correction factor to be applied to local time stamps.

Given the system configuration addressed previously and clock function templates, we can define a feasible technique for automating the resolution of distributed clocks. Using this technique, a simple program, customized to the user's environment, will be automatically generated whose sole purpose is to determine the correction factor to be applied to each processor's clock values. In this clock measurement system, each processor will be assigned a process designed to communicate with the other processes residing on separate processors. The behavior of this clock correction measurement operates as follows. One designated process must determine the difference between its clock and the clocks on all other processors in the system. It will first exchange a set of messages of constant-size with each of the remaining processes. Elapsed times of each exchange will be measured and used to compute the message delay time. Next, a series of messages will be sent back to the designated process by each of the others containing its local time. The first process will use its knowledge of message delay and its current time to calculate the difference of the other clocks. This correction factor will be applied during post-processing of timing trace data. For systems where all processors are not able to communicate directly, the algorithm previously described will exist as a hierarchy of processes based on a spanning tree of the processor-to-processor communication links. When each layer of measurement processes complete their interchange, delay factors will be propagated up to the next level, until they are computed for each processor. It is important to note that the correction calculation algorithm should be run prior to the program being analyzed as well as afterwards. For example, independent execution of clock measurements will be used to calculate anomalies such as clock drift. If the compensation data is collected again immediately following the execution of the same test case, the deviations in the individual compensation values can be used to determine each clocks rate of drift to further enhance the accuracy of the time stamps in relation to the global system timeline.

The overall structure of the clock compensation process required to minimize clock errors in the collected trace data is illustrated in Figure 5.2.3.3-1 and Figure 5.2.3.3-2. The collection of clock compensation data is performed immediately prior to the execution of a test. After the test case is executed, the clock compensation data is used in the correlation of time stamped events to a global system timeline with minimal clock error.

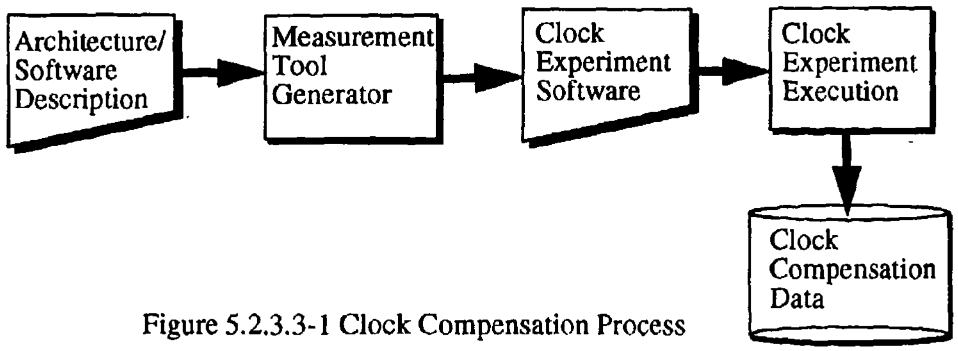

Architecture/ Software Description
Measurement Tool Generator
Clock Experiment Software
Clock Experiment Execution
Clock Compensation Data

Figure 5.2.3.3-1 Clock Compensation Process

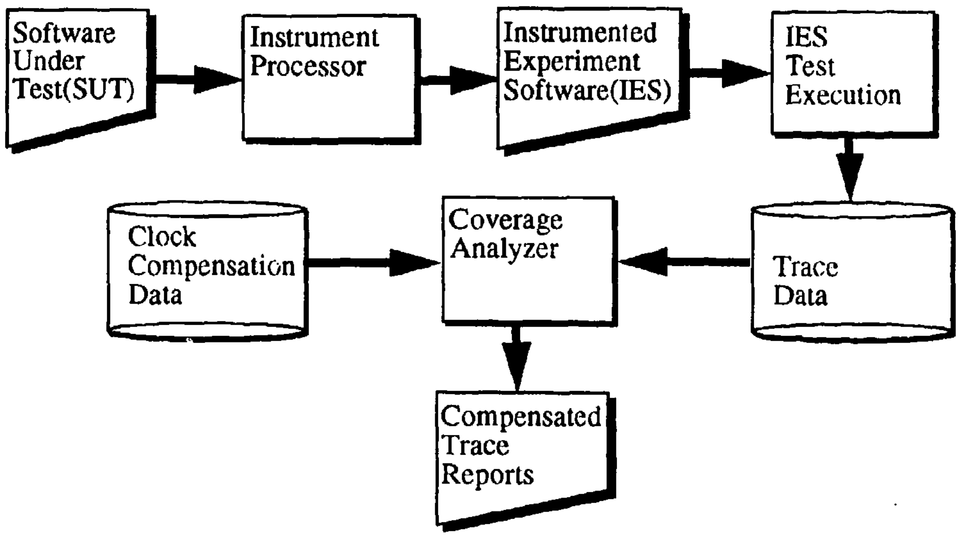


Figure 5.2.3.3-2 Clock Error Compensation of Time-Stamped Trace Data

5.2.3.3.1 Correction of Distributed Local Clocks

Once the ETADS tool is able to distinguish which processor each process or task executes on, each of those individual processor clocks must be related to one relative system global time.

ETADS is designed to address the testing issues related to real-time distributed systems. In some cases, real-time systems have time-triggered events. Often, these systems are running on distributed systems with synchronized clocks. In other words, the distributed system itself periodically synchronizes each of the local clocks so they are all the same. Although these clocks will not always be identical, due to such factors as drift, the system itself does as much as possible to keep them synchronized. In such a system, distributed time-stamps are already related to each other so no post-test calibration of time stamps is required. Although this is a trivial solution to the problem of accessing distributed clocks, it must not be overlooked by the ETADS tool. If a system is known to be running in such an environment, ETADS should use the time-stamps without corrections.

At the opposite extreme, a real-time system may be event-triggered rather than time-triggered. In this case, the distributed clocks will not be calibrated with respect to one another at all. In order to produce useful timing analysis reports, ETADS must include a utility that is capable of relating the time-stamps gathered from distributed clocks to each other.

In order to translate and instrument programs in order to trace the execution of most events, ETADS must already have static knowledge about the processors involved, how they communicate and how to query clocks. Therefore, ETADS should be able to generate a clock calibration program to measure the difference between one clock and all others in the system. A simple outline of the algorithm is given in section 5.2.3.3.2. It is important to note that this program can be run prior to the system under test to collect information about clock differences. It can also be run after the system test and the results of the two runs could be used to estimate clock drifts over the duration of the test run. Note that variations in system load interfering with interprocess communication should average out of the timing difference measurement. Based on the range of values computed in any given run, one could also place a statistical confidence level on the accuracy of the clock adjustments.

5.2.3.3.2 Clock Difference Algorithm

One processor must be designated as the master and it must be able to communicate with all other processors in the system. This will not be an issue with PVM systems. However, if this is not the case, ETADS can restructure the algorithm to collect timing differences between sets of processors that are able to communicate with each other, then send the results to one master processor as long as it is able to communicate with at least one in the group. If the master processor cannot communicate at least indirectly with all other processors in this fashion, then it is impossible to relate the clocks to each other. This should not be a problem, however, when performing software testing of a distributed system. If two programs are unable to communicate or interact in anyway, then they actually are not part of the same distributed system.

The algorithm can be rewritten as a hierarchy of processes based on a spanning tree of the processor to processor communication paths. At the bottom are simple slave tasks. When they complete, the next level of tasks send the lower level data up to their masters prior to beginning their slave procedure, and so on until all timing differences are propagated up to one master processor. Although this will require more sophisticated correction mechanisms, the implementation of such mechanisms is entirely feasible.

This procedure should be avoided if all processors can communicate directly, because each level of data has to be corrected to the next level clock, thus producing results less accurate than one level of refinement. On the other hand, this procedure may provide more accurate timing corrections for systems where near-neighbor processors are allowed to communicate via the master-slave hierarchy, and far-neighbors do not calibrate with each other directly but with another processor in common. Again spanning trees of processor organization would be a useful input to the ETADS as a part of the architecture specification.

Another important assumption is that the slave knows the unit of measurement used by the master processor. If it is different, the slave can correct the time values to the proper unit before comparing to its own clock value. It must also send the data back to the master using the appropriate time unit.

The following algorithm can be augmented to collect statistics on the variance of clock differences computed. These statistics, such as standard deviation, can be used to generate a confidence factor for the accuracy of the corrected time-stamps.

Master Process / **Slave Process**

```
Master Process                              Slave Process

for each slave processor i do
   wait for slave to be ready               signal master that I am ready
   /*determine delay incurred to
   exchange message with slave*/
   loop N times:                            loop N times:
      time_a = read_my_clock()
      send time_a to slave i                      recv time_a from master
                                                  temp = read_my_clock()
      recv time_a from slave i                    send time_a to master
      time_b = read_my_clock()
      diff = (time_b - time_a) / 2
      total = total + diff
   end loop                                 end loop

   msg_delay = total / N
   send msg_delay to slave i                recv msg_delay from master

   /*compute difference between
   two clocks*/
   loop N times:                            loop N times:
      time_a = read_my_clock()
      send time_a to slave i                      recv time_a from master
                                                  my_time = read_my_clock
                                                  adjust = time_a + delay
                                                  diff = my_time - adjust
                                                  total = total + diff
      recv ready from slave i                     send ready to master
   end loop                                 end loop

                                            avg_diff = total/N
   recv avg_clock_diff from                 send avg_diff to master
                  slave i
```

```
                                    end Slave Process
   record avg_clock_diff for
                  processor i
                  in trace file

end for each slave processor
end Master Process
```

The first loop N is used to compute the message delay incurred when the master sends a message containing its current time value to the slave. The master computes the average delay time of a one-way clock time message. The second loop N is used to repeatedly measure current time and send it to the slave process. The slave updates the current time of the master with respect to the delay incurred by sending the message, then compares the corrected time to its own current time when the message was received. The result reflects the difference between the two clocks. Again, these steps are performed repeatedly and the results are averaged in order to minimize error due to fluctuations in current system load.

Once all average clock differences are known, any time stamps gathered can be adjusted as if it were measured on the master processor's clock, thus creating a system time reference based on one processor's clock. Timing analysis data can then be post-normalized to start at a system time of 0 and presented to the user in reports.

5.3 Deterministic Execution Testing

The traditional approach to testing the functionality of a program is to execute it with selected test cases and compare the results with those expected. If the test case identifies errors, the software may first be re-executed using the same test case to find the error and later executed to determine if changes have corrected the fault. However, for distributed and concurrent software, rerunning the program using the same test case is not guaranteed to produce the same results. Due to variations in processing rates combined with the possibility of data or message races, distributed and concurrent software systems often exhibit non-deterministic behavior, i.e. a program may behave differently in each test run even though the same input is used when executing it. For distributed software, it is necessary to provide the facility to force the recurrence of the same software behavior in each test execution. This is called deterministic execution, sometimes referred to as deterministic replay, or simply replay.

To enforce deterministic replay, software instruments are used to collect data at the occurrence of important process events during the execution of a test. These data are processed after test case execution to create a sequence of process interaction events that characterize the behavior of the execution. Any event that may be involved in non-deterministic behavior, e.g. an accept or message receive, must be captured. Forcing the same sequence of such events, when combined with the same input as the original program test execution, will force the replay execution to behave in the same manner as before. There are different approaches for controlling deterministic replay depending on the type of process interaction events that may occur. However, the approaches are similar in that they all require the tracing of process interactions.

5.3.1 Objectives and Benefits

The top-level objective of research in this area is to provide the means for executing distributed real-time software in a deterministic manner by eliminating or controlling the sources of non-determinism in the SUT. By providing such services, ETADS will significantly reduce the difficulties encountered in testing distributed real-time software, thus increasing the achievable productivity in software testing and thereby improving the software quality.

ETADS will provide services to observe and record the behavior of the SUT for a specific test case. Given a record of the test case behavior (i.e. in the form of a trace file), ETADS will provide services to execute the SUT once again and derive precisely the same behavior with each execution.

As an unanticipated benefit associated with deterministic replay, the technique can facilitate mutation testing in the distributed real-time environment. Integrated support can be provided for construction of mutants of the SUT, after which the deterministic replay service can be used to control the manner in which test cases are executed.

In summary, benefits from deterministic execution testing include: 1) facilitate reproducible testing in a complex environment, 2) facilitate software debugging, 3) support for mutation testing in the distributed real-time environment.

5.3.2 Technical Issues

The deterministic execution problem is divided into two separate, but intimately related, subproblems: 1) test case execution with observation and recording SUT behavior, 2) test case execution with forced sequencing of task interaction events (TIE).

In addressing this problem domain, the following technical issues and questions must be addressed and satisfactorily resolved:

- What minimal subset of task interaction events (TIE) and data must be recorded to completely and uniquely identify the system state throughout the life span of the test case?
- What data collection techniques (i.e. software probes) are required to gather the event data?
- What forms of software probes, and probe processing software, are required to force the replay of a particular sequence of events?
- Define and document the technical limitations of the replay technique with respect to reatime software behavior.
- Replay Initial Conditions And Inputs
- Capture And Replay All Inputs
- Clock References By SUT
- Non-Determinism in External System Inputs

5.3.3 Research Results

The general approach in developing deterministic replay techniques relies heavily on system modeling and analysis at this point in SBIR research development. Realizing that this is a pioneering technology area for which there has been no documented success in the open literature (i.e. only limited success in restricted domains has been reported), the first concern in this development was to define a feasible strategy for subsequent implementation under a Phase II SBIR effort. The Phase I research effort has been pursued with the recognition that the funding and schedule constraints would not permit development of a working prototype as this is a large technical area and is only one of several areas pursued in Phase I.

The Phase I effort focused on several interrelated task activities including: definition of data collection requirements, evaluation of experiment perturbation effects due to software probe insertion for data collection, evaluation of experiment perturbation effects due to software probe insertion for replay, and definition of requirements for automated data collection.

Short of actual implementation, the evaluation of the feasibility of each ETADS capability is supported through a constructive argument. If we can identify a feasible approach to implement a solution for each individual technical issue, then we are more confident in the feasibility of the integrated approach for deterministic replay.

5.3.3.1 Replay in Rendezvous-Like Languages

The 1991 paper "Debugging Concurrent Ada Programs by Deterministic Execution", describes the issues required to force a replay based on a given sequence of synchronization events. In a language where most interprocess synchronization occurs via named rendezvous (such as Ada), the order in which interprocess events occur can be reproduced.

The fundamental idea for replaying an interprocess event (such as a rendezvous) is that a "guardian process" must control the order of entries to a process by other processes. This is achieved by requiring the caller to acquire (wait for) permission from the guardian before executing the actual entry call existing in the un-instrumented version of the program. The guardian process must have knowledge (i.e. input) of the order that entry requests can be made. It then uses this input ordering (TIE-sequence) to identify the next caller it must wait to grant permission to. The input to a process' guardian is the TIE-sequence containing the sequence of events in which the guarded process is the receiving task (i.e. called task). Therefore, a partial ordering of events with respect to each process is sufficient to force replay.

This is a simplified view of how deterministic replay must hold off callers until it is appropriate for them to rendezvous with the guarded task. There are several other details that must be dealt with respect to the called task, but these details are given in the Tai paper mentioned above. Also, the authors of the paper also provide the mechanisms for transforming more complicated Ada-like non-deterministic events such as select statements in order to force replay. These techniques can be adapted to similar rendezvous based languages. See the referenced paper for the details. This existing tool, although limited to Ada code, is evidence of the feasibility of automating the deterministic execution of concurrent programs.

ETADS must provide necessary mechanisms to extend the protocol to deal with multiple versions of the same task (such as arrays of tasks). This can be accomplished by using a combination of a unique task id (generated at translation time) with an indexing scheme to identify the current instance of a task. Then one could create an array of guardians, one for each instance of the task being guarded. Also the input TIE-sequences must include the index, so the appropriate guard instance will reference the appropriate sequence of events.

The following is a simple example of an entry call transformed to obtain permission to rendezvous from a process guarding entries into Process A (original Code is shown in **bold**). This caller, identified by my_id, will suspend at the guardian call until permission is granted. The guardian process uses my_id to determine when to allow that process to request the next rendezvous. After the permission is received, the sender can request the actual desired rendezvous with Process A.

```
Guardian_of_PROCESS_A.Get_Permission(my_id); -- my_id is sender's task id
PROCESS_A.EntryCallStmt....;
```

5.3.3.2 Replay In Synchronous Message Passing Systems

A general message passing system which uses synchronous communication can be replayed using the same model as for Ada-like languages. In this context, synchronous communication means that the sender blocks until its message is received by the destination process (similar to a rendezvous). Also, the receiver blocks until a requested message arrives.

In this case, the order that messages are sent can be controlled by granting the sender permission to send the next message. Again, a guardian process can be generated to intercept permission requests from tasks desiring to send messages. The guardian for a particular process grants permission to the next appropriate sender, based on an input TIE-sequence of events.

The following is an outline of the protocol used to replay a synchronous send message event from process A to process B. Original code is shown in **bold**. In this and future pseudocode examples, send represents a synchronous send, nsend represents an asynchronous send, recv represents a synchronous receive and nrecv represents an asynchronous receive.

```
-- process A must request         -- Process B has a guard
-- permission before sending      -- inserted that grants
-- actual message to B            -- permission to the next
                                  -- valid sender

Process A                         Process B

send(B, permissionreq);           call guard();
recv(B, permission);              recv appropriate message;
send(B, message);                 ...
...

                                  -- B's guard determines who
                                  -- the next valid sender is from
                                  -- B's TIE sequence.
                                  guard()
                                     senderid = next valid sender
                                     recv(senderid, permissionreq);
                                     send(senderid, permission);
```

The above approach is only appropriate for point-to-point synchronous message passing mechanisms. It is not adequate for a broadcast mechanism, nor for asynchronous send or receives. In order to grant a broadcast permission, the sender would have to broadcast the permission request, then wait for all processes involved to send permission back to the requesting sender. The sender would then require a "guard" procedure that indicates (based on TIE-sequence data) how many permissions to wait for. Still, this is only applicable to synchronous (blocking) broadcasts, not asynchronous broadcasts.

5.3.3.3 Replay In Asynchronous Message Passing Systems

Both of the previous approaches depend on controlling the order in which senders (callers) are allowed to send messages to (rendezvous with) a particular destination task. However, this approach is not appropriate in a system where sends are asynchronous (nonblocking). Consider the following simple scenario between three processes A, B and C. The statement **nsend** is used to represent a nonblocking send. Assume **recv** blocks until message arrival.

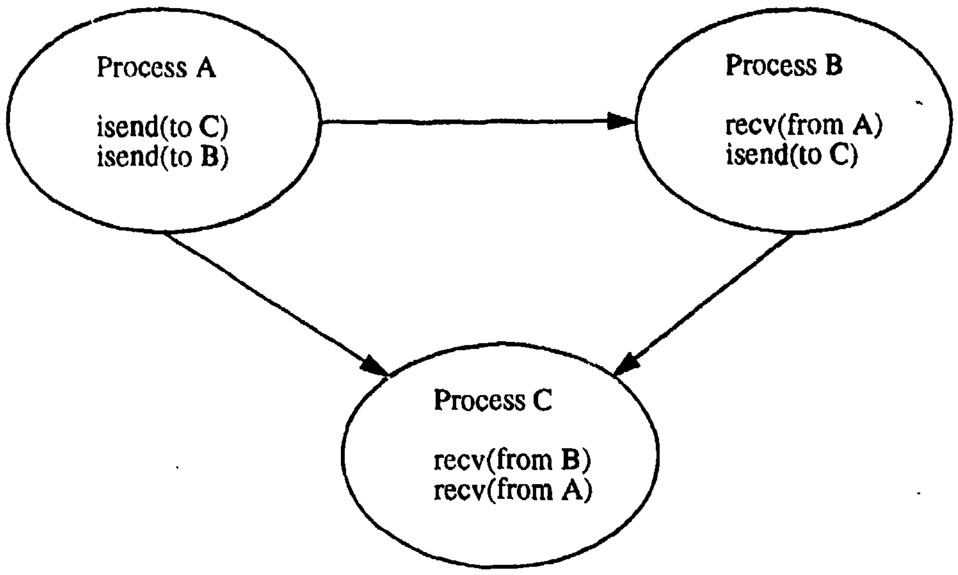


Figure 5.3.3.3-1 Asynchronous Communications Example

Consider what could happen in the above scenario if one attempted to force synchronization on the asynchronous sends of process A. Process A would first request permission to send to process C. However, C's guard would not grant permission to A until after it granted permission and received a message from B. Because A is blocked waiting for permission, its message to B could not be sent, creating a deadlock in the transformed system. This simple example shows that it is not appropriate to force nonblocking sends to be blocking.

This problem is very significant in that many distributed environments use asynchronous message sending and receiving functions. If ETADS is to address the needs of realistic environments, one can expect to encounter this problem. As an example, consider the popular PVM (Parallel Virtual Machine) language extensions to C and Fortran developed at ORNL. The PVM libraries provide users with a uniform language with which to develop distributed programs on distributed machines. PVM is popular with both researchers and developers as indicated by the existence of a USENET news group dedicated to discussion about PVM and its uses in applications. PVM is an example of a popular distributed programming environment that uses nonblocking sends.

Also, even if the user is developing distributed processes that communicate using network communication functions, one can expect these functions to be capable of implementing nonblocking sends.

5.3.3.4 Approaches to Forcing Replay for Asynchronous Sends

Nondeterminism in systems implemented with communication based interprocess activities arises when a process may receive messages in a non-fixed order from other processes. A race condition is said to exist if messages from the other processes may arrive in different orders in different test runs due to some uncontrollable scheduling activity. In order to replay a certain sequence of events, one must control the order in which those messages are accepted by the receiver.

Phase I research has identified several existing techniques for producing deterministic replay of a program, most notably those of Tai and Carver. However, these solutions require synchronous send/receive activities. ETADS proposes to address the shortcomings of these approaches with respect to asynchronous interprocess communication.

5.3.3.4.1 Strategy 1: Using System Generated Task ID's (tids)

As stated earlier, to replay a specific sequence of interprocess communication events, one must be able to control the order in which messages are accepted by the receiving process. In a system using asynchronous sends, you cannot force the sender to wait for permission to send. Therefore, whenever a process needs to receive a message, it must be guaranteed to be the correct one. In an environment like PVM, nondeterminism arises when a process could potentially receive a message from multiple sources. One way to control from which source you receive a message is to specify the source process' tid as a parameter to the receive operation. A **tid** in this context means a task or process identification generated by the run-time scheduler. Tids are used to refer to specific processes with in the program. This allows the receiver to specify the source it expects to receive from next (based on a TIE-sequence from a previous run).

The first approach uses this idea. When a receive is encountered, the guard function must supply the appropriate **tid** of the process from which the next message is to be accepted. The remaining parameters of the original receive call are unaffected.

Although the approach is fairly simple, the main difficulty lies in determining what the appropriate tid value is. Remember that in a system like PVM, tids are generated dynamically at run time by the scheduler, and therefore the tid value corresponding to each process will change from run to run. The user has no control over what the tid values will be. Also, it is necessary to replace the tid parameter in the code with the one supplied by the guard so that the receive is limited to accepting a message from exactly one valid process at a time, as in the following example segment:

```
Process XYZ
   ...
   next_tid = guard_get_tid(); <- guard returns tid that next
                                  message should come from
   recv(next_tid, message,...); <- transformed receive
   ...
```

Originally, the guard function has as its input, not a TIE-sequence containing tids, but a sequence containing logical process identifiers (*pid*), representing tasks translated into ETADS. Each process in the system must have a unique pid. The guard must map these pids into their corresponding actual tid. In order to accomplish this at run-time, the guard must query a task registry process in order to determine the current tid value of a particular process. This task registry process maintains the correspondence between tids and pids as processes are activated in the program. It will be created by ETADS and is also responsible for generating unique pids for all processes at run-time (see

Figure 5.3.3.4.1-1). This task registry object is described further in the section on collecting necessary trace data to produce TIE-sequences.

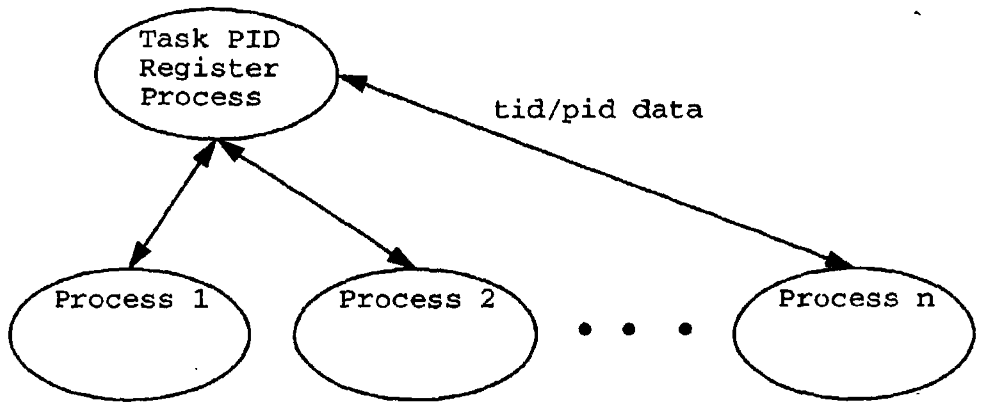


Figure 5.3.3.4.1-Process Identification and Registration

A major drawback to this approach is the fact that the task id registry process could become a communications bottleneck. Whenever a process needs to know the tid of some other process, it must communicate with the task registry process to obtain it. However, if the amount of interprocess communication is small relative to the overall processing of the system, then the delay incurred during replay should be tolerable.

An attractive feature of this method for replaying asynchronous communication is that it is language independent. Even if the distributed system communicates via network communication libraries rather than a language extension such as PVM, there still must be some manner for the processes involved to uniquely identify sender and receiver processes for routing communication to the proper one. As long as there is a mechanism for collecting process tids and using them in communication functions, this approach will work.

Additionally, the creation of a task registry process, capable of generating (and repeating) a unique set of pids, is already required to produce meaningful trace data for a distributed system of processes. Each process must be assigned a unique pid with which its trace data can be identified. The basics of this process is discussed in section 5.3.3.4.2.

Below is a simple outline of one guardian function for a particular process which determines the tid of the other process involved in the next event. Each transformed process will have as input a TIE-sequence that must be replayed. For communication based synchronization this will mainly consist of receive events. One might think of this sequence as a list of event types and the associated pid of the originator of the event (such as a sender of a message). Each guard function must have access to this data. Guard function calls are inserted into the transformed program, to facilitate the replay of certain events.

```
Guardian_A Data: - data for guardian of process A
  TIE-sequence for process A
  mapping of ETADS pids to system run-time tids

Guardian_A Functions: - set of functions that can be
                                          inserted into process A

-- this sample guard function queries the central process id
-- registry task to get the current tid associated with a
-- logical pid
function guard_get_tid()
  look up next TIE-sequence event to get pid of sender
  search mapping for tid corresponding to pid
  if (tid not found) then
    msg_type = lookup request
    message.pid = next event sender's pid
    send (to registry task, msg_type, message)
    recv (from registry task, msg_type, message)
    tid = message.tid
    record tid in mapping
  end if
  return tid
end function
```

Depending on the number of different types of events being reproduced, there may be several other types of guard procedures. For instance, if this were a synchronous send environment, the sender processes must request permission to send from a guard at the receiver processes.

5.3.3.4.2 Generating Trace Data for synchronization events

The creation of a task registry process, capable of generating (and repeating) a unique set of pids, is required to produce meaningful trace data for a distributed system of processes. Each process must be assigned a unique pid with which its trace data can be identified. This is also necessary to support the strategy described above to control message reception based on run-time task identifications.

These pids cannot be completely generated statically since duplicate instances of tasks could be created at run time. At best ETADS will be able to assign unique pids to each type of task in the system, but at run-time, these pids must be augmented to indicate which instance of a task each process is. Therefore, a pid generated by the task registry process could be a combination of a predetermined pid and an index for each instance of that process. The task registry process must also maintain a mapping of these pids to the run-time system's generated tids, so that ETADS pids can be related to actual tids when necessary (as needed for replay).

A process need only register with this task once upon its inception. A process can then tag its generated trace data with its unique pid. Either the task register or each individual process should also place into the trace output which system tid maps to its ETADS pid. This will be necessary to relate interprocess communications across multiple trace files. Basically, when a process sends a message to another, it knows the tid of the destination, but not the unique pid of the destination. If the send is to be properly matched to its destination process, the tid used in the send event can later be matched to the proper pid by post-processing the trace file.

Note that for languages such as Ada the sender's pid can be added as a parameter to the accept. Thus the receiver can record the actual pid of the sender. In this case, no special processing is required to relate tid to pid.

One drawback is that system generated tids might be reused (as in PVM). The system will need to detect when a tid becomes associated with a new process.

As mentioned in strategy 1 above for replay, a task registration process such as this will be useful for replay. During replay, a process knows which logical pid it should receive the next message from (based on the input TIE-sequence). The task registry will enable it to map that pid to an actual current tid within the running system.

Below is a simple outline of the functionality of a basic task registry process. The mapping array might be represented as an array of lists, one per ETADS task type. Each task type can then have a list of instances mapping task pid and index into a unique pid. The executing task's tid can also be stored in this array for future lookup or trace generation.

```
Task Register Process
   mapping = an array of lists.
             Each entry of a list maps a pid to a tid.
   message = a complete message. A message will include any data
            (such as pid) pertinent to the message type.

loop
   recv (from any process, msg_type, message);
   if (msg_type is a registry request) then
      Search mapping[message.pid] to determine next index value
      unique_pid = combination of message.pid and index
      Create new mapping record with unique_pid and sender's tid
      msg_type = reply
      message.unique_pid = unique_pid
      send(to requesting process, msg_type, message);
   else if (msg_type is a lookup request) then
      Search mapping to find tid corresponding to message.pid
```

```
        msg_type = reply
        message.tid = tid
        send(to requesting process, msg_type, message);
else
   - handle other message types such as abort or generate trace
end loop
end task register process
```

5.3.3.4.3 Strategy 2: Using Message Tags

Another approach to controlling incoming messages is to use message typing facilities if they exist. Once a trace has been analyzed, TIE-sequences could be generated where senders and receivers tag messages with a unique type. Then receivers could wait for a message of the expected type rather than figuring out the tid of the sender.

Although this approach may eliminate the communication overhead incurred by gathering tid data from a centralized register process, it will be more difficult to implement in practise. First, trace data will have to contain the message types used in the original run of the program. Then, ETADS must generate a new set of message types that do not use any of the message types recorded in the trace. Finally, user code may be likely to use the value of message types in the code after a message is retrieved. If this is the case, then the appropriate message type must be reproduced at the point where it is needed. These requirements may hinder the ability to use this approach. However, if the overhead of a centralized register process is deemed too great, it may be worth investigating the potential use of this approach.

5.3.3.5 Approaches to forcing replay for asynchronous receives

Whether t!· ɔender is synchronous or not, asynchronous receives should not be a problem for replay. An asynchronous receive is one that receives the next message if it is there, and if a message is not available control returns to the following statements. An asynchronous receive will have to be transformed into the body of an if statement that replaces it with a synchronous receive. Then, the TIE-sequence data must indicate if the receive is successful. If so, the replay will block at the transformed receive until the appropriate message arrives. If not, the transformed segment of code should set the appropriate return values so the remainder of the un-transformed code executes as intended. The data in the TIE-sequence must contain the required return data. The guard function that replaces the asynchronous receive statement can be written with a parameter list that mimics the receive parameter list. This way, the user code can be instrumented easily to allow the appropriate variables to be passed to the new function. For example, if the original asynchronous receive statement is:

```
error = nrecv(tid, msg_type, message)
```

The transformed segment resembles the following (bold indicates original code):

```
error = guard_next_recv_occurs(tid, msg_type, message,
                                      recv_occurs)
if (recv_occurs) then
  recv_tid = guard_get_tid()
  recv(recv_tid, msg_type, message)
else
  null
```

Also, note that a timed-receive (i.e. a receive that delays for some amount of time before unblocking if the receive is unsuccessful) can be transformed into an if statement that either waits for the message using a synchronous receive or delays for the specified amount of time if the receive was unsuccessful.

5.3.3.6 Combining approaches

In the ETADS Phase II development, OTI plans to incorporate support for both synchronous and asynchronous communication environments. This will enable ETADS to be more useful for a larger class of distributed programming environments. It should be noted that current ETADS research does not address the issues of low-level, shared memory synchronization such as semaphores and monitors. Approaches have been developed for these scenarios, but it must be recognized that they can be computationally expensive (inserting semaphores may incur busy-waiting in some implementations). This philosophy during Phase I research is consistent with our placement of priority on researching approaches that are mainly used in a large-scale, distributed memory environment.

5.3.3.7 Definition of synchronization events

The following is a tentative list of task interaction events (TIE) that must be traced by each process to reproduce an execution. Traces of these events (when augmented with time stamps) can also be used in other analysis areas of the tool (such as timing analysis reports).

1a. start synchronous send
1b. end synchronous send
2a. start asynchronous send
2b. end asynchronous send
3a. begin synchronous receive
3b. end synchronous receive
4a. begin asynchronous receive
4b. end asynchronous receive with message
4c. end asynchronous receive without message
5a. begin timed receive
5b. end successful timed receive
5c. end unsuccessful timed receive (timed out)

Note that it is not sufficient to merely record the fact that a send or receive starts. The completion of such an event must also be recorded. If an event does not end or ends with an error, the TIE-sequence produced from this trace must reflect and reproduce these types of events.

5.3.3.8 Deterministic execution aids software testing

The conventional approach to testing a sequential program is to execute it with each selected test input then compare the test results with the expected results. If the test input uncovers errors in the software under test, the program may be rerun using the same input to find the causes of the error. However, for concurrent software, simply rerunning the program using the same test input is not sufficient. Concurrent programs often are capable of non-deterministic behavior. In other words, even though the program is executed using the same input, it may behave differently in each test run. Therefore, in order to support adequate software testing mechanisms the same behavior must recur as in each test run. This is the goal of deterministic execution.

However, deterministic execution techniques are not limited to using the same test input and the same set of TIE events to force the same execution repeatedly. The techniques can be applied in other areas in an attempt to uncover more run-time faults.

One application is to use the same TIE-sequence from one test case, but alter the input to see if that particular execution sequence is still valid. This is sometimes referred to as testing the feasibility of the TIE-sequence. If the execution succeeds using the new input, the sequence is said to be feasible for that input. If not, the sequence is said to be infeasible. Testing sequences for feasibility under different input conditions may provide the user with useful information about the run-time behavior of the program.

Another area in which deterministic execution can be used is in the support of mutation analysis. After various TIE-sequences are collected for a program, the user can apply mutation techniques to the program. He or she can then run the mutated version of the program using the same input and syn-sequence combinations uncovered during previous tests. The details of mutation analysis are supplied in Section 5.5.

It is important to note that when forcing replay, exactly the same inputs must be used. This indicates that all input to the system must be duplicated. This could be very difficult in that all external events must be the same as they are part of the input of the program that determines behavior. While users may have some control over these events, it may be infeasible to duplicate the input in some cases. For example, one area in which this can be expected to occur is a system involving time-triggered events. If a system causes events to happen based on reading the clock (i.e. other than time-outs which are more easily handled), then all clock readings must be captured and simulated as part of the replay. One can see that this could require an enormous amount of trace data, especially for large systems.

5.3.3.9 Deterministic Replay Process Summary

A work flow process has been defined for the deterministic replay technique and is illustrated below in Figure 5.3.3.11-1.

The user's original source code is run through the Translator of the ETADS tool which performs several processing steps including: 1) construct an abstract representation of the original source and store it in the repository, and 2) save a copy of the original source in the repository for later recall.

In order to execute a test case and collect trace data, the user must "Instrument" the original source code (i.e. insert software probes at specific locations in the source text). The probe insertion process is automated under the control of the "Instrumentor" component of the ETADS tool. The user accesses the Instrumentor through the normal ETADS graphical user interface and requests that the source code be instrumented using a specific strategy. In this case, the user selects the strategy for data collection for deterministic replay. Running the Instrumentor generates a set of instrumented software, for which the complete collection of software is said to form the program P1.

Execution of the program P1 in the target environment generates a set of one or more trace files. One trace file is generated for each execution of a test case.

The trace files are processed by the "Event Analyzer" after SUT has been executed one or more times. In processing the event data from a trace file, the Event Analyzer references the abstractions contained in the repository to correlate event reports to specific locations within the SUT.

A second program P2 is generated based on a particular trace file. The program P2 is instrumented using a second instrumentation strategy whose goal is to force a particular sequence of events. Thus, the instrumentation contained within P2 is distinctly different from that contained in P1.

In addition to generating P2 for each trace file, the event analyzer must generate a set of guardian software to force event sequencing. The guardian will be a custom set of software configured for a specific test case and must be linked with the corresponding program P2.

Subsequent execution of P2 on the target environment will generated precisely the same sequence of events as was recorded for P1.

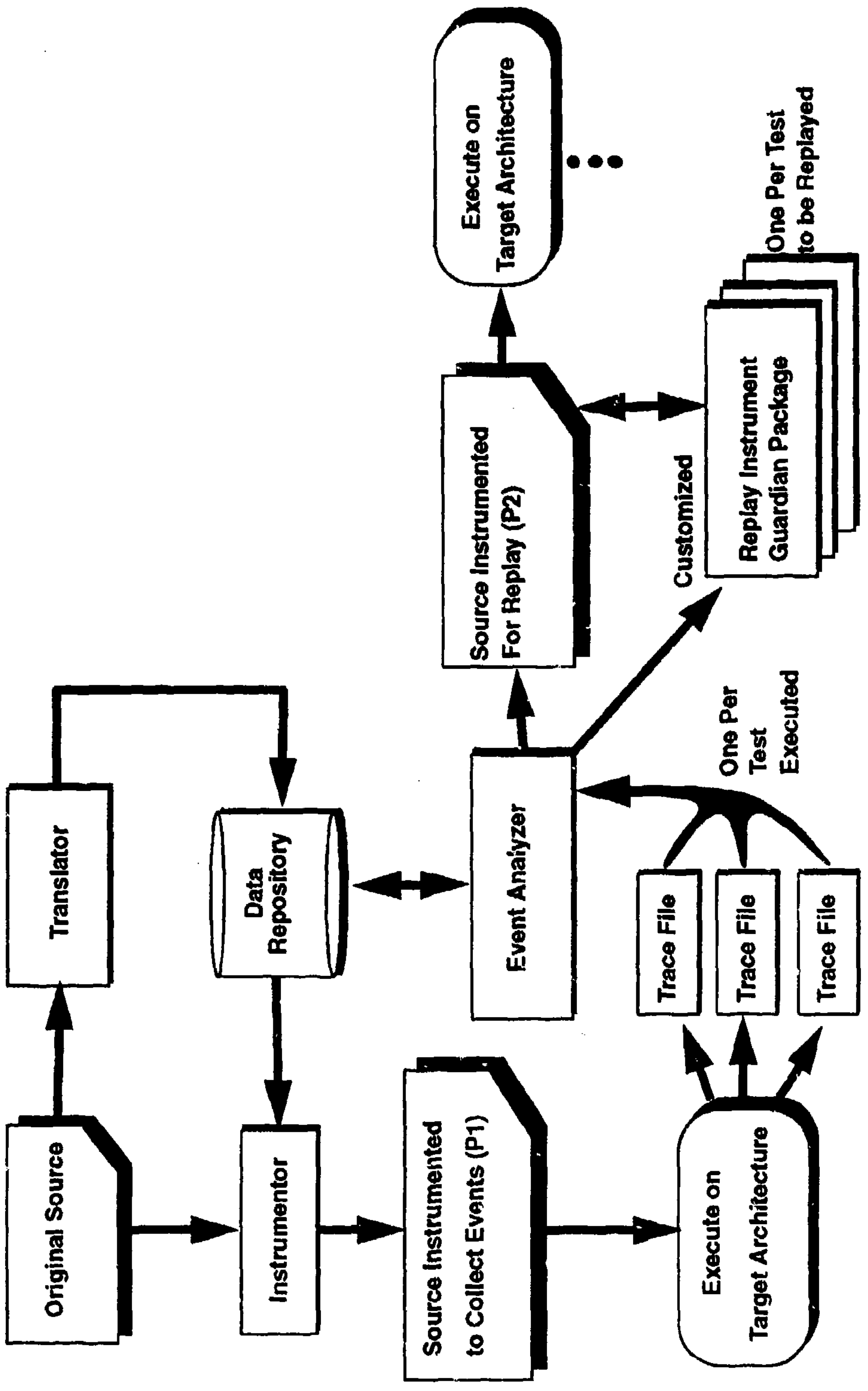

Original Source
Translator
Data Repository
Instrumentor
Source Instrumented to Collect Events (P1)
Event Analyzer
Execute on Target Architecture
Trace File
Trace File
Trace File
One Per Test Executed
Source Instrumented For Replay (P2)
Customized
Replay Instrument Guardian Package
One Per Test to be Replayed
Execute on Target Architecture
•••

Figure 5.3.3.9-1 Deterministic Execution Replay Work Flow

5.4 Dependency Analysis

When testing a large software system of any kind, it is important to generate an overview of the behavior of the total system. A higher-level abstraction of the software under test is the first step in determining where more detailed analysis should be focused. Dependency analysis of distributed software provides this higher-level view by focusing on the impact that concurrency has on the overall system. Areas included in this analysis are reports of interprocess communication, process creation activities, and quantitative measures of the complexity added by the presence of concurrency.

5.4.1 Objectives and Benefits

The overall objective of dependency analysis can be summarized as the intention to enhance the understanding of complex software behavior through the application of advanced visualization techniques for showing process dependencies. Just as sequential software can be described as having a static structure that is expressible in terms of a flow chart, concurrent software can be described as having static structural relationships that may be expressed graphically. One natural view of concurrent software is one in which parent-child relationships are prominently displayed. Some desirable graphical views include:

- interprocess dependency diagrams
- Processor/Process diagrams
- Shared data/Process dependency diagrams

Dependency analysis can be performed in two different modes of operation:

- static analysis mode, and
- dynamic analysis mode.

The static analysis operations are performed without any actual execution information and may be performed prior to the first execution. The focus is to identify the static parent-child process relationships (i.e. parent process A creates child process B). The results of static analysis are dependent to some degree on the success achieved in data flow analysis for the given language. Data flow analysis performance will depend on how process identifiers are manipulated within the SUT. Other goals of the static analysis mode are the static identification of patterns of message traffic between parent-child processes and sibling processes. Further analysis for some operating environments such as with PVM may benefit from the identification of process group relationships (i.e. clusters of related processes as defined by PVM and MPI). In contrast, dynamic analysis mode requires the actual execution of the SUT and the recording of trace data. Post execution analysis of the trace data may reveal additional parent-child process relationships that were not identifiable statically, may identify dynamic patterns of message traffic and may identify membership and communications patterns with respect to process groups.

For large scale distributed real-time systems, there may be a large number of processes involved in the overall computation. As the size of the system increases, it becomes more difficult to identify and verify the relationships between various processes whose very existence may be of a transient nature. ETADS dependency analysis focuses on mitigating this complexity by providing automated visualization tools. Visualization may reveal unexpected interactions or interactions that should occur but for some reason do not occur.

5.4.2 Technical Issues

The implementation of automated ETADS support for dependency analysis requires the consideration and satisfactory resolution of the following technical issues and/or questions:

- What are the possible, statically identifiable process relations based on the existing code?
- What are the observed process relations based on execution of the code?
- Observation of process performance in an unobtrusive manner
- Static data flow analysis for process identifier variables

5.4.3 Research Results

The research approach is based on the development of paper models of the proposed system and the manual analysis of the system properties prior to full scale development. Evaluating the feasibility of individual elements of the approach is dependent on manual examination of all relevant ETADS design elements for consistency and completeness. In addition, comparison of the design elements of the proposed technical approach to similar techniques incorporated in other systems indicates feasibility.

Integration with existing SADCA technology can take advantage of existing data flow analysis techniques. The data flow analysis must be extended to identify and trace variables used to record process identifiers. For those process identifiers whose use can be traced statically, the associated processes may be uniquely identified in the static dependency graphs. Due to the nature of this analysis, it can be expected there will be wide variation in the ability to statically identify process relationships depending on the language environment in combination with the selection of programming techniques applied in the SUT. For example, if the SUT is coded in C using a custom communications library with many arrays and/or pointers referencing process identifiers, then static analysis may be unable to statically identify many processes relationships. In contrast, if PVM/C is the language of choice and unique scalar variables are used to record process identifier (i.e. tid - task identifier) values, static analysis may be able to construct a complete process relationship graph.

5.4.3.1 Process Creation Report

A process creation diagram will show parent/child relationships between processes in the system. Both static and dynamic views can be generated as shown in Figure 5.4.3.1-1. The static view will show potential parent/child relationships detected during static analysis of the system. The dynamic view will show process creation relationships that actually happ..ns during a test execution. These reports enhance the user's understanding of process creation at a higher-level of granularity.

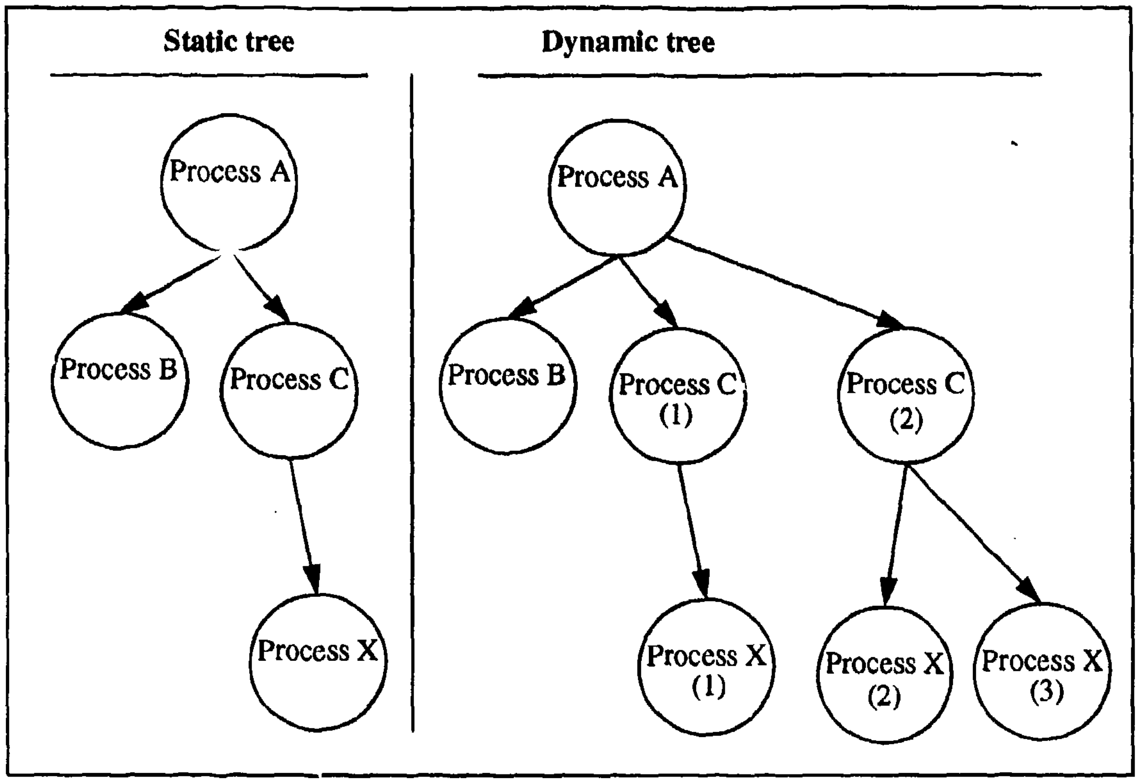


Figure 5.4.3.1-1 Static vs. Dynamic Dependency Diagram

Additional information may be retrieved upon demand from the dynamic dependency diagram using standard ETADS popup menu operations. The user may point the cursor to a particular node of the dependency diagram and with the click of the mouse button be presented with a menu of additional options. For example, one option could be to present a detailed process interactions diagram (i.e. an example of such a diagram is presented below). Alternatively, the user might select a view which shows the process to processor mapping.

5.4.3.2 Process Interaction Report

A process interaction report will present the process interactions at the event level (e.g. communications level). The static report will indicate potential process interactions such as pairs of processes that could be involved in each particular communication event, while a dynamic version will show actual process interactions as they occur. This view could be further augmented with the timing analysis data to indicate when events occurred during the overall system time.

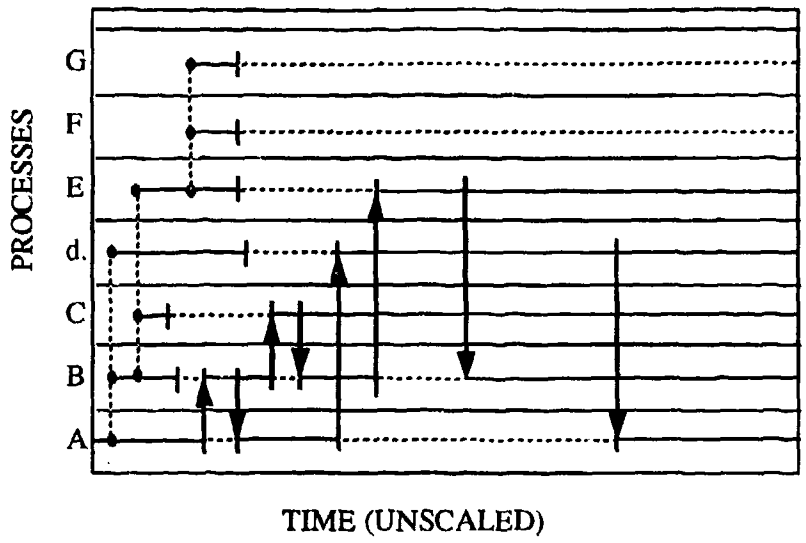


Figure 5.4.3.2-1 Process Interactions Diagram

5.4.3.3 Concurrency Metrics

Useful statically computable sequential complexity metrics, Halstead's Software Science and McCabe's Cyclomatic Complexity, have been extended in research to account for additional quantitative measure of complexity due to concurrency.

5.4.3.3.1 Extending Halstead's Software Science

The original Halstead measures are based only on the syntax of the program text, operators and operands, without considering the semantics of the application software. Extensions to Halstead's metrics are proposed to measure the impact of concurrency on the software. These measures which were originally proposed for Ada code (and could easily be extended to other languages) are called Ada-Multitasking-Real-time (AMR) metrics. Operators and operands associated with multitasking (e.g. task specifications, task and activations), inter-task communication, non-deterministic constructs, and real-time constraints are counted much like Halstead's original tokens and used in formulas to compute AMR predicted effort and predicted bugs.

5.4.3.3.2 Extending McCabe's Cyclomatic Complexity

McCabe's approach is based on graph theory and defines complexity in terms of the cyclomatic number. This number is derived solely from the control flow graph and thus is independent of any particular language. Control flow for concurrent software involves interprocess interactions as well as traditional sequential constructs, and the extension to McCabe's measure considers this added complexity. Because Petri Nets are a graphical representation of processes and their interactions, the new metric will measure the cyclomatic complexity of a system composed of Petri Nets. Because any concurrent system can be modeled with nets, this approach remains independent of any particular language. The metric is computed in the same manner as McCabe's for traditional control flow graphs by counting decision paths, resulting in a single quantitative value for the entire system of processes.

5.5 Mutation Analysis

5.5.1 Objectives and Benefits

The immediate objective of mutation analysis is to generate a set of mutant application programs, each of which has one carefully selected syntax change applied to it. One can then run the mutant versions with the test data for which the original program correctly executes. If the output from a mutant version differs from the original version, then that particular test case is said to have killed the mutant. A perfect test set would thus be able to kill all mutants generated from a program. The goal is to develop a suite of test cases to kill as many mutants as possible.

The objective of mutation analysis is to improve, or at least facilitate the measurement of, the software reliability. Mutation analysis provides a mechanism for evaluating the completeness of a suite of test cases in terms of its ability to detect defects in the software. Application of the method provides objective measurements against which the requirement for additional test cases can be quantified in comparison to the software reliability requirements.

5.5.2 Technical Issues

The integration of mutation analysis services into the ETADS tool environment requires the consideration of the following issues/questions and the development of a reasonable solution approach.

- How should be the criteria for selecting mutations to be applied to real-time distributed software?
- What degree of automated support is desirable and/or required to support mutation testing for the ETADS problem domain?

5.5.3 Research Results

The general approach for developing mutation analysis support involved the following steps: Define data requirements for the mutation analysis features and reports, define data collection requirements and the steps required to collect the requisite data in the ETADS environment, compare ETADS data collection and user interface requirements to existing SADCA capabilities.

The support for mutation analysis defined during the Phase I SBIR effort includes a combination of automated support for experiment construction and user guided insertion of mutations via the graphical user interface. The experiment construction capability is combined with the ETADS Deterministic Replay facilities to provide a unique software testing capability for the distributed real-time software domain.

The mutation analysis support developed for the ETADS architecture takes maximum advantage of the existing SADCA architecture to yield maximum benefit in terms of new software testing functionality while minimizing the effort required to acquire the technology.

Evaluation of the feasibility of integrating mutation analysis into ETADS is based on the manual comparison and analysis of the data requirements and processing requirements anticipated for mutation analysis in comparison to existing capabilities available in the SADCA architecture. The result of this analysis is the evaluation of the prospects for implementation as being highly feasible within the limitations defined for ETADS support in the near term which includes automated support for user directed insertion of mutations. In the long term, an additional ETADS support module could be developed to provide automated creation and insertion of "high-payoff" mutants.

5.5.3.1 Mutation Analysis Background

In the report *Program Mutation: An Approach to Software Testing*, DeMillo defines **program mutation** as "a method of assessing the quality of computer program test data. Test data is said to be adequate if a program behaves correctly on the data but incorrect programs fail. A **mutation score** assesses how close a given test is to being adequate."

The main idea behind mutation analysis is to generate a set of mutant programs, then run them with the test data for which the original program runs correctly. If the output from a mutant version differs from the original version, then that particular test set is said to have "killed" the mutant. The goal is to develop a suite of test cases to kill as many mutants as possible. (A perfect test set would thus be able to kill all mutants generated from a program.)

It is important to realize that mutating a program means applying one change to it, such as changing a relational operator or a variable in an expression. Applying every possible mutation to a program creates a quite impractical large set of mutant programs. Each one must be run and its output compared with the original. For any significant program, this is very expensive, time-consuming operation. As a result, people have tried to develop good sets of mutations, then attempt to develop test suites that achieve a high percentage of kills. (For more details on mutation theory, see the bibliography references.) Although time-consuming, this approach to software testing is useful for very critical software, and is often applied during unit testing, rather than integration testing.

In order for a ETADS tool to be capable of automatically generating mutant programs, the translators would have to collect and store not only variable references in equations, but also operator types and locations. As seen with SADCA's static analysis databases, the amount of storage needed just for variable references can grow to be very large. This would be even worse for operator uses. Of course, if a tool were to only implement certain kinds of mutations such as relational operator changes, the amount of stored data is much less. More research into mutation testing is indicated to decide whether sufficient motivation exists to implement in a multilingual tool with automated support for mutant generation. Current research alludes to the fact that mutants are usually generated differently for different languages and for different application environments. It may be difficult to support enough variety to satisfy all users across a broad spectrum. In the interim, automated support for user-guided insertion of mutations followed by automated generation of the mutant version of the SUT is a feasible alternative to complete automation.

The described approach in section 5.5.1 is referred to as strong mutation. Another approach, weak mutation, has been developed to reduce the amount of computation necessary to test mutant programs. This approach generates the set of changes to the program. An interpreter executes the original program up to a certain mutated segment of the code, called a component. The program state is saved. Then the normal component is executed, and the new correct state is saved. The interpreter then rolls back to the saved state prior to the component, and inserts the mutant component. As each mutant component is executed, its state is compared to the proper program state. This effectively prevents recomputation over and over of prior program activities that are not affected by changes. However, this method only determines when a program state is different than the original. It does not check the final outcome of the test. The authors indicate that this approach works well in many cases and have defined several types of components suitable for weak mutation. See the paper by Offutt and Lee for more details.

Although this approach reduces the computation time required, it is not automatable by a ETADS tool. This is due to the fact that the tool itself would have to symbolically execute the program under test.

5.5.3.2 An Approach For Mutation Testing of Concurrent Software

Carver states that "mutation-based testing has been applied to sequential software; however, problems are encountered when it is applied to concurrent programs. These problems are a product of the nondeterminism inherent in the executions of concurrent programs."

Carver advocates combining deterministic execution with mutation testing. First the program can be executed to collect a set of TIE-sequences for the given input data. This results in a set of TIE-sequences and possibly a set of valid output, due to the non-deterministic behavior of the concurrent software. Once a program's TIE-sequences are collected, they can be used to force deterministic replay.

The next step is to create the desired set of mutant programs and run them with the replay input for the original program. Then the output and stopping conditions of the mutant can be examined to see if it differs from the original for the test input. Carver calls his approach deterministic execution mutation testing (DEMT), and formally describes the method in his paper.

By incorporating a deterministic execution replay facility into ETADS, the tool would be close to supporting mutation analysis for concurrent software. Rather than attempt to generate mutant programs automatically, the tool can be extended rather simply to allow the user to specify changes themselves. This would be done by allowing the user to select a code component in the user interface, and indicate which text to change. An instrumentation command can be generated to handle the change when the code is instrumented later.

Note that the user will have many options once mutant program versions are specified. He could choose to instrument the code to produce only the new mutant, or the instrumentation could be combined with replay instrumentation and TIE-sequence. Also, the mutant can be combined with coverage tracing or timing probes. Any combination could feasibly be allowed.

This simple compromise will allow users in many environments to use mutation-based testing if they wish. They will have control over how their mutant programs are generated. The important feature is that deterministic replay will facilitate the correct execution of mutant code during testing.

5.5.3.3 ETADS User Assertions

It is important to note that ETADS can utilize the same facilities for inserting mutation changes to allow the user to insert their own instruments or assertions. The user will not be allowed to change tasking components, but could choose to insert their own probes most other places. The only restriction is that the user is responsible for ensuring that their additions are syntactically correct.

It is notable that the ETADS Deterministic Replay service supports this as well, as long as the user doesn't change how the tasking will be performed. For example, the user may want to rerun the program for a given test case but with his own probes inserted for debugging purposes or for general purpose performance data collection.

5.6 Control Flow Testing

Control flow testing is designed to evaluate the degree to which control flow paths have been exercised by one or more test cases drawn from the suite of test cases.

5.6.1 Objectives and Benefits

Control flow metrics for sequential software aid users in the definition of test sets that can adequately exercise the software and give a certain degree of confidence that the software will perform correctly. They also provide a criteria for identifying when to cease testing as well as a formal way of quantifying test case coverage. The current SADCA test tool provides control flow information for sequential software in the form of coverage reports that indicate which statements have been exercised as well as the outcome of decision/condition points in the code. Coverage analysis also quantifies the percentage of the software that has been executed (i.e. covered) during testing.

It has been demonstrated that coverage analysis tools can effectively increase software reliability through their consistent application to critical software. Such tools provide the means to achieve the required visibility into the software testing process such that objective management decisions can be made to improve the performance with respect to testing. It is extremely important to extend this class of tool support to include distributed real-time systems. The need for automation in this area is clear when one considers that no tool currently available on the market addresses this need.

5.6.2 Technical Issues

The development of specific techniques for coverage analysis features requires the detailed consideration of each of the following technical issues:

- What are appropriate representation mechanisms for reporting control flow coverage for distributed real-time software?
- Definition of specific task interaction events (TIE) to be observed and recorded
- Definition of software instruments to record TIE's
- Evaluation of alternative techniques for coverage analysis of concurrency state space

5.6.3 Research Results

The research approach for development of coverage analysis techniques has relied on using our experience in designing coverage analysis tools for sequential software and considering the additional complications inherent to the distributed real-time environment. In comparison to the supporting techniques for sequential instrumentation and reporting of coverage results, we reviewed the data requirements for the new domain and constructed the new set of data collection requirements. In this technology area, there clearly exists a large degree of commonality between existing SADCA technology and the required ETADS technology.

Comparison of SADCA and ETADS requirements indicates that implementation of the ETADS services is entirely feasible.

5.6.3.1 Task Interaction Coverage Analysis

The traditional control flow coverage analysis for sequential software can be extended to distributed software in two ways. First, traditional statement, decision and condition coverage analysis will be applied at the process level. Since a process is very similar to a sequential program, coverage reports are vital aids in determining which sections of processes are executed by test cases. Second, coverage analysis will be extended to treat distributed computing constructs, such as communications events, as special forms of control flow, much like decision/condition constructs are analyzed in sequential code. Dynamic coverage analysis will then identify which processes are affected by the execution of certain events as well as indicating the outcome of events, such as the success or failure of a message receive request.

5.6.3.2 Alternative Coverage Metrics

With respect to sequential control flow analysis, dynamic coverage information will identify what events have occurred as well as the percentage of code or events actually executed. What dynamic coverage analysis, for either sequential or distributed software, does not compute is the actual number of possible paths through the software that have been executed. Therefore, to support the goal of providing more complete control flow reports, OTI examined a set of control flow metrics proposed for concurrent software.

The group of alternative metrics considered include concurrency state coverage, state transition coverage and synchronization coverage. Each of these metrics relies on the concurrency state graph used by static concurrency analysis (see section 5.1). Each metric indicates the extent to which the graph has been exercised during testing. The implementation of these metrics suffer from the same drawbacks as static concurrency testing in that the concurrency state graph can grow too large such that it is infeasible to enumerate all of the possible states in the state space.

Efforts to optimize the concurrency state graph for static analysis should also consider the impact on control flow testing. Also, one should consider decomposing the system into smaller subsystems whose concurrency state graphs are manageable in size, and analyze each subsystem separately.

5.6.3.3 Example Coverage Graphics

One form of extension of the coverage analysis graphs currently supported by SADCA can be accomplished by defining new elements (i.e. Components) for the "Common Language" supported by the SADCA repository. The new components will each have its own predefined graphic symbol by which the user will be acquainted with the originating task interaction statement. An example segment of a flow chart containing such elements is illustrated below in Figure 5.6.3.3-1. For presentation of coverage results, color coding is used to indicate which parts of the graph have been exercised by one or more test cases. The color mapping is indicated in Table 5.6.3.3-1.

Table 5.6.3.3-1 Color Code Mapping

COLOR	COVERAGE
Red	Not Hit
Green	Hit
Yellow	Partial Coverage

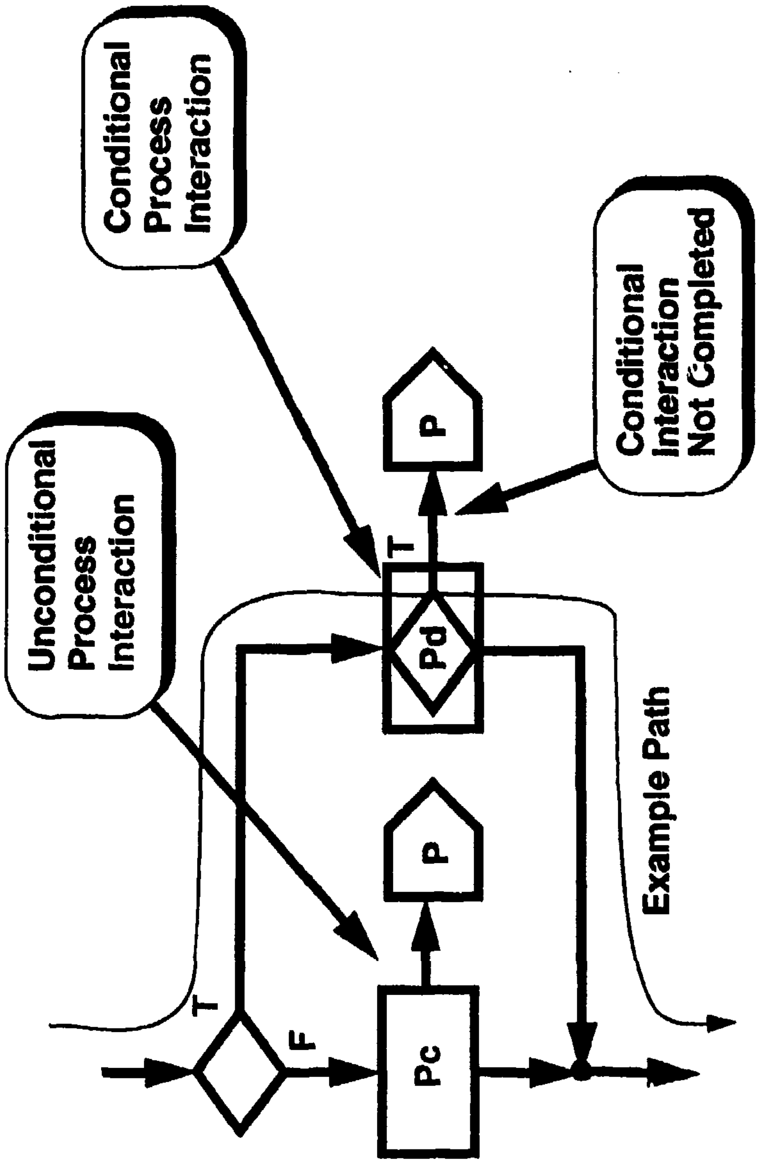

Conditional Process Interaction
Conditional Interaction Not Completed
Unconditional Process Interaction
P
P
T
Pd
Example Path
T
F
Pc

Figure 5.6.3.3-1 Example Coverage Chart With TIE Components

5.7 Network Communications Testing

5.7.1 Objectives and Benefits

The primary objective is to support monitoring of inter-task network communications while minimizing the degree to which the act of observing the behavior affects the process performance. In contrast to other ETADS services that are available to focus attention on a variety of other aspects of interprocess communications, this service is provided to provide visibility into the software behavior from the perspective of network communications. Whereas, other views may become cluttered with other types of event data, this class of report focuses on the network message traffic.

Information of interest in this area include timing relationships of message events. Given trace data of such events, informative displays can be presented to the user to describe the system performance and provide the basis for comparison against the expected behavior.

5.7.2 Technical Issues

The principal technical issues related to network communications testing are those shown below.

- What data must be collected?
- What graphical displays are appropriate for review of network traffic data?
- What kind of special software probes are required to implement the recording of message contents?

In general, we identify all message events as being part of the required data input to network communications testing. Usually, each message event should also include a time-stamp whenever possible to facilitate timing diagrams of message data.

5.7.3 Research Results

The network communications testing techniques make extensive use of the existing SADCA structure. Additional graphic output services must be defined.

The types of reports defined in this category of analysis include histogram message frequency (e.g. by process, message type, etc.). In addition, graphs can be generated on the basis of message event time (e.g. by message type, etc.).

An alternative technique has also been defined for recording message content and dumping the recorded data into an output stream.

All of the identified techniques have been evaluated in comparison to previous SADCA implementation experience and found to be feasible.

5.7.3.1 Network Communications Reports

Distributed processes are most likely to interact and exchange information via message passing. As such, there is a need to understand the patterns of communication occurring during program execution. Understanding these patterns can aid the user in focusing on potential problem areas related to communication. Also, studying run-time message patterns can provide valuable insight into potential bottleneck areas where performance may degrade in a real-time environment.

Network communication testing provides direct visibility into the communications between processes executing either on the same or different hosts. It provides an alternative view at a higher level of abstraction focusing narrowly on the interprocess communications. Communication event instrumentation technology, already required for deterministic replay, combined with time stamp data and system architecture information provides the requisite data for construction of a variety of informative communication performance graphs. Such displays facilitate human comprehension of complex real-time performance data.

OTI has identified a variety of communication event data, which when traced, provide input for several types of displays. Network communication graphs provide a detailed view of the message activity ongoing in the system during a user selected time interval. For example, they can illustrate the frequency of each message type sent or received by a particular process or processor as directed by the user. These data could also be combined for all message types to show combined message activity for each process or processor. Sample displays are shown in the Figure 5.7.3.1-1.

OTI has identified a variety of communication event data which if traced provide input for many types of displays. Network communication graphs provide a detailed view of the message activity ongoing in the system during a user selected time interval. For example, they can illustrate the frequency of each message type sent or received by a particular process or processor as directed by the user. This data could also be combined for all message types to show combined message activity for each process or processor. Sample displays are shown in the following figures.

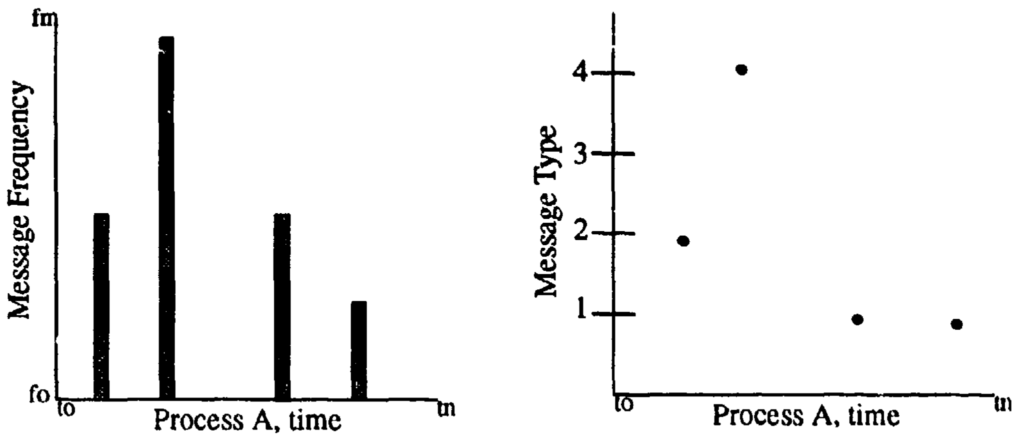


Figure 5.7.3.1-1 Sample Network Communications Graphics

The first display indicates the frequency a particular message type is sent (or received) by a particular process. This display can also be computed for individual processors as well as processes. The second display illustrates a process's activity for all message types. Again, this report can be computed at the processor level as well as for individual processes, providing views at different levels of granularity. Since there may be many other combinations of message type/process activity, the possibilities for displaying the data is open-ended and may be supported by user interface tools to let the user select from a wide range of options.

5.7.3.2 Message Content Tracing

Another objective for the Phase I SBIR research was to evaluate the feasibility of providing automated support to trace the content of particular messages. The concept has been found to be feasible and a technique has been defined to implementing the feature within certain predefined limitations as described below.

It is entirely feasible to implement a strategy for recording the contents of a message and dumping the recorded data into an output stream. However, the utility of the collected data blocks is outside the bounds of the ETADS tool. ETADS can provide the means for collecting such data through the normal ETADS graphical user interface and can direct the data blocks to an output stream as specified by the user. However, such a data collection process is comparable to the collection of telemetry data and requires the existence of an external tool to interpret the recorded data.

5.8 Problem Tracking and Reporting

5.8.1 Objectives and Benefits

Understanding the cost-effectiveness of testing techniques is an important aspect of software testing.

Because it is important to relate the history of using a tool to the productivity gainea from applying selected techniques, OTI feels that a problem tracking and reporting facility is an important addition to any modern test tool.

For each testing technique applied to software testing, a complete history of the number and type of errors found should be maintained.

To achieve this, a database can be developed that tracks the types and numbers of software errors found along with the technique that was instrumental in identifying each fault.

Providing a problem tracking and reporting tool as part of ETADS will encourage users to record information related to detected faults, as well as estimates of the cost/benefit of applying specific tool techniques, at the time that testing is being performed.

Histories in the database can be analyzed to determine which techniques or combinations of techniques were most beneficial in the testing process and can be used as an objective source of data for improving the software testing process.

Finally, if the tool is readily available as an integrated part of the software testing tool environment, users will not be tempted to delay recording such data.

5.8.2 Technical Issues

The technical issues related to integrating problem tracking and reporting services into the ETADS environment are listed below.

- What critical mass of problem data is required to support the generation of useful statistics with regard to tool utilization
- Database management of problem reports
- Statistical evaluation and reporting of problem report data

5.8.3 Research Results

The problem tracking and reporting services is defined as an integrated part of the ETADS environment to facilitate its use in a timely fashion. The service can be described as consisting of the following major pieces: 1) a database management system (DBMS) and 2) a statistics calculation and reporting service. The required DBMS services are relatively simple storage and retrieval capabilities with some predefined query services (e.g. construct a list of current problem reports).

Integration with existing SADCA DBMS services is anticipated to be entirely feasible with no significant complications since the information contained in the existing database and the additions required for problem tracking and reporting form a disjoint set.

5.9 Distributed Clock Synchronization/Compensation

Distributed systems are, by definition, distributed physically and interconnected through some form of communication subsystem. As a consequence of the physical separation, it is common for the various machines comprising the network to have no central clock available. Thus, the system operates with the presence of a significant degree of clock error that must be carefully managed and accounted for in all subsequent calculations. The same clock limitation applies to the calculations performed by any software testing tools designed to operate in this environment.

5.9.1 Objectives and Benefits

The overall objective of this technology area is to compensate for the limitations imposed on timing measurements due to distributed clock errors. Many of the primary, user oriented technologies described in earlier chapters of this final report provide analysis results whose accuracy is to some degree influenced by the accuracy of the time stamps applied to the trace data.

Ultimately, one would like to have a perfect global clock from which time stamps would be drawn and attached to each time stamped event. However, this scenario is generally not the case in the distributed environment, and therefore steps must be taken to compensate for clock errors. In general, the process for using ETADS must include the steps for clock compensation since it is relatively difficult for the average user to independently evaluate the clock characteristics. This is a function for which an automated service is best applied.

The benefit of performing this analysis should not be understated and bears repeating. Many of the other ETADS analysis features is to some degree dependent on the accuracy of the time stamp information.

5.9.2 Technical Issues

The fundamental technical issues driving the need for clock compensation include:

- distributed clock errors
    - some degree of error
    - some degree of drift
- cannot assume control over the distributed clocks
    - may lack sufficient user privilege
    - may lack hardware support
    - may lack software support

5.9.3 Research Results

Extensive literature review of previous results generated in this area exposed techniques that can be adapted to the ETADS environment. Summary description of the details of the techniques planned for ETADS was presented as a part of the discussion of Timing Analysis in Chapter 5.2 and will not be repeated here.

5.10 Time-Stamp Generation

5.10.1 Objectives and Benefits

The Time-Stamp Generation technical area is not an objective in itself, rather it is an intermediate objective whose resolution approach impacts other ETADS technical objectives. The objective in this area is to define the strategy for associating time stamp information with specific recorded events.

The primary benefits associated with this objective are concerned with resource utilization. The collection of time stamps requires a finite amount of CPU time to generate each time stamp, a finite amount of primary memory resource to record each time stamp prior to output, and a finite amount of secondary storage for long term storage. In addition, the utilization of network I/O bandwidth to transfer time stamped trace data must not be neglected. It is very important to optimize the utilization of these resources in order to minimize the impact on the SUT behavior and to reduce secondary storage resource consumption.

5.10.2 Technical Issues

The implementation of time stamp generation services into the ETADS tool environment requires the consideration of the following issues/questions and the development of a reasonable solution approach.

- How can CPU utilization be minimized?
- How can primary memory utilization be minimized?
- How can network I/O bandwidth consumption be minimized?
- How can secondary storage consumption be minimized?
- How can these issues, and their resolution approaches be balanced?

5.10.3 Research Results

Enumeration of the possibilities with regard to time-stamp utilization shows the following small set of possibilities and their associated implications:

- Time-Stamp on: Every Event (i.e. Every Software Instrument)
- Time-Stamp on: Subset of Events (e.g. Subroutine Entry, Message Send, etc.)
- Time-Stamp on: Task Interaction Events Only
- Time-Stamp on: No Events

5.10.3.1 Every Event Time-Stamps

As one of the possible alternatives, this approach is not attractive. The application of time stamps to all events would result in massive data storage requirement for no apparent benefit. Very few of the events that would be recorded need to have time stamps in order to provide useful information to the user. This option also introduces the maximum perturbation due to CPU utilization.

5.10.3.2 Task Interaction Event Time-Stamps

This option associates time stamps only with the Task Interaction Events (TIE). Each TIE implies some finite amount of communication between two processes, each of which otherwise operates on its own timeline with no regard for the timing of other processes. The association of time stamps with each TIE permits the post execution construction of a partially order set of events. The set of events is only partially ordered since the recorded events within a process that do not have time stamps can only be placed in relative ordering according to the sequence in which they were recorded. This operating mode is generally sufficient for reporting the operations within a task as the time between events is not a factor in determining the behavior (i.e. under the assumption that there are no other performance effects, such as shared memory, that are being neglected). In the absence of shared memory, which is generally the case for distributed systems, the time stamps on TIE's are a necessary and sufficient condition for constructing the partially order set.

5.10.3.3 No Time-Stamps

This option eliminates time stamps from consideration and only records events in sequence as they are encountered. For synchronous communications languages, such as Ada, this is mode of operation is sufficient to construct a partial ordering of all events. However, insufficient information is contained in the data set to produce a total ordering. In contrast, for asynchronous environments such as that provided with PVM, or the general case of distributed computing environment, one cannot construct a partial ordering of all events. It is only possible to construct a partial ordering on a process-by-process basis as there is no interprocess timing information present in the data set.

5.11 Minimization of Probe Effects

5.11.1 Objectives and Benefits

The insertion of software probes to collect trace data must be performed in a manner such that the required observability into the test execution is achieved with the minimum possible perturbation of the experiment.

The target environment is real-time distributed software for which the behavior of the SUT is typically influenced by time (i.e. non-deterministic behavior is an issue). The incorporation of any modification of the SUT has the potential of disturbing the timing relationships and thus changing the observed behavior of the software. This tendency must be minimized by minimizing the influence of the probe insertions.

5.11.2 Technical Issues

The implementation of techniques to minimize probe effects in the ETADS tool environment requires the consideration of the following issues/questions and the development of a reasonable solution approach.

- Define a minimum set of probes for the user specified experiment objective
- Define software probe processing routines that minimize the impact with respect to time

5.11.3 Research Results

The language-based approach using software probes has been proven successful in the sequential software domain through the development and application of SADCA and METAsoft. Some degree of performance optimization techniques have been implemented in both SADCA and METAsoft. These techniques can be readily integrated into ETADS as well.

Software testing requires that one must gather data relative to the quality, correctness, structure, etc. of the software in both static and dynamic states. This implies a reliance on data collection mechanisms, and as in any application of measurement theory, the intrusiveness of the collection instrument is of concern. Special care must be taken when the software under test must support real-time system applications. For static testing, where the software is not executed, but simply scanned and analyzed for anomalies, standards, metrics, etc., there are no detrimental probe effects. The concern arises, due to slowdown caused by intrusive measurement devices when dynamic testing is performed on executing software. When these instrumentation devices are code segments for collecting and storing measurement data (i.e. must also be executed in-line with the applications software), there is a critical reason to exercise caution.

5.11.3.1 Hardware Instrumentation

An obvious non-intrusive measurement device is hardware instrumentation. This technique will not change the timing behavior of an executing program. However the special hardware for supporting this approach is target machine dependent, very often environment specific, typically very expensive and in many cases technically infeasible. These deficiencies prevent its serious consideration for supporting a system/language independent software test environment.

5.11.3.2 Kernel Instrumentation

Another data collection approach investigated was based on detecting and recording message interactions using functions built into a modified system kernel. Trace data would be collected by inserting software probes into the operating system at locations that either process interrupts or provide support for user programs. However, this approach would rely on changing the environment of the system on which the user's software would be executing, and furthermore it would be necessary to customize the kernel on every machine in a distributed system, which may have different operating systems. This deficiency and the associated cost (estimated to be some order of magnitude more than implementing the same function outside the kernel) also prevents its serious consideration for supporting a system/language independent software test environment.

An example implementation of this approach was documented by Miller et al in the open literature describing the IPS-2 parallel program system. The system was implemented on two operation systems. An approach such as this could probably be used for an environment such as PVM where all machines are using the same underlying PVM mechanisms for interprocess activities. However, even in this case, there will still be interference suffered from the insertion of probes in the underlying system.

5.11.3.3 Selectable Instrumentation Levels

In order to minimize intrusiveness and unwanted trace data, the instrumentation process must be optimized. This will be accomplished by allowing the user to select from a set of predefined instrumentation levels (such as process communication, function calls, statement coverage, etc.) as well as providing user designated instrumentation across the topology of the program to be tested. Although this cannot completely eliminate the effects caused by probe insertion into the software, flexible sets of instruments and designated instrumentation provide the user extensive control over the process which is also highly automatable.

5.11.3.4 Trace Data Optimization

In a distributed system there should be a trace file for each processor which should be collected and merged after test completion as a post processing task. Two methods for minimizing slowdown due to insertion of data trace operations in software are storing intermediate trace data in memory and limiting the transfer points at which trace data is transferred to disk. Periodic flushing of memory to permanent storage will be restricted to certain program event points, thereby reducing the overhead caused by disk I/O. Such event points are characterized as periods of low computation such as synchronous communication events or procedure returns. Traditional data compression techniques shall be utilized to reduce the volume of trace data generated

5.11.3.5 Summary

The first two methods described above are not recommended for further development in ETADS since they are highly system dependent approaches. In order to produce a language and system independent tool, we have no choice but to insert software probes into the program under test. We must take advantage of the optimization techniques described to limit the impact of probe insertion as best we can.

An important issue related to the collection of trace data is the distributed nature of trace files. For efficiency, there should be a trace file for each processor involved in the program. Trace files can be collected after the program test is completed, and merged during post-processing. One potential problem is that in a distributed environment, all processors may not have access to permanent I/O facilities. If this is the case, trace data must be transferred to another processor to be stored. This could have a significant impact on program timing and the implementation approach must be carefully considered.

5.12 Trace Data Compression

5.12.1 Objectives and Benefits

The Trace Data Compression technical area is not an objective in itself, rather it is an intermediate objective whose resolution approach impacts other ETADS technical objectives. The objective in this area is to define the strategy for reducing resource consumption. Resources targeted by this technology includes: primary memory, network I/O bandwidth, and secondary storage. The benefit of this technology is that it reduces resource utilization in these areas. The cost is that it tends to increase the resource utilization with respect to CPU usage.

5.12.2 Technical Issues

The implementation of Trace Data Compression services into the ETADS tool environment requires the consideration of the following issues/questions and the development of a reasonable solution approach.

- Define fast data compression techniques
- Reorganize trace record structures to reduce size as much as possible

5.12.3 Research Results

A general strategy for accomplishing trace data compression within the framework of the ETADS architecture has been defined and is described below.

The ETADS compression strategy consists of a two stage pipeline and utilized a combination of techniques. In the first stage of the pipeline, ETADS development must include a reorganization of the trace data records to minimize the amount of overhead associated with each piece of trace information. In the second stage of the pipeline, ETADS can utilize common compression algorithms to perform the compression of the data stream. Several alternatives exist in the public domain which can be directly integrated with minimal effort. Figure 5.12.3-1 illustrates this pipeline structure.

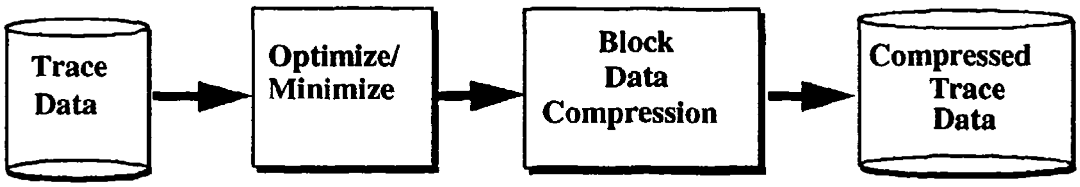


Figure 5.12.3-1 ETADS Trace Data Compression Pipeline

Optimization/minimization of instrument data within trace records may take advantage of a variety of candidate techniques including: 1) using relative time-stamps wherever possible, 2) avoidance of unnecessary time-stamps (e.g. partial ordered set of events requires fewer time stamps), 3) intelligent trace record packaging.

5.13 Trace File Management

5.13.1 Objectives and Benefits

The Trace File Management technical area is another area in which it is not an objective in itself, rather it is an intermediate objective whose resolution approach impacts other ETADS technical objectives. The objective in this area is to define a uniform strategy for managing the creation and transfer of trace files.

The mode of operation of ETADS in performing dynamic analysis activities of various types requires the automated analysis of trace data following each test case. In the sequential domain, the identification and management of trace files is relatively uncomplicated. However, in the distributed system environment, special attention must be placed on the definition of a strategy to coordinate the collation of trace data from a wide variety of network hosts into a coherent package of trace data.

It is important that this issue be resolved in a manner that minimizes the explicit human involvement in the trace file management process. This will serve to maximize the use of computer automation for the mundane housekeeping activities of dynamic analysis and allow the ETADS user to focus on higher-level software testing issues. In addition, the minimization of manual steps in the process reduces the opportunity for human error to be introduced into the process.

5.13.2 Technical Issues

The implementation of Trace File Management services into the ETADS tool environment requires the consideration of the following issues/questions and the development of a reasonable solution approach.

- Distributed file system access
- File system availability
- Permanent storage limitations
- Trace File transfer operations
- Trace File merging and sorting operations

The target operating environment can generally be expected to include a heterogenous distributed file system. However, there is not guarantee that all nodes in the network of computing resources will have a local file system that can be accessed by the ETADS instrument processing routines. Thus, the design and implementation of the instrument processing software should incorporate facilities to determine the availability of resources on the local host and apply the most appropriate data transfer mechanisms to report trace data back to the ETADS host.

5.13.3 Research Results

The approach in development of trace file management techniques has been to enumerate the alternative strategies and compare characteristics with the approach already implemented in SADCA. This serves to highlight the commonality between ETADS versus SADCA approaches and identify candidate reuse technology. It also serves as a sound basis for comparison against a proven technology.

The concurrent nature of the distributed software environment yields a new phenomena in trace file construction and management: multiple trace data streams to be merged and sorted at the ETADS host. In the sequential domain, with only a single thread of control, a single trace file is generated by a given test case execution. In contrast, the distributed domain has multiple threads of control and may produce multiple trace files, each of which must in some way be merged and sorted to produce a single global view of the experiment behavior.

The following strategies for trace file management have been identified: 1) multiple trace file creation with post-experiment merge-sort operation, 2) multiple network I/O streams to a single trace file with post-experiment merge-sort operation, and 3) hybrid of approach 1 and approach 2.

The first approach is applicable when each node of the distributed network has its own local file system. During test case execution, each process is responsible for writing its on trace file, insuring the proper open-close file operations and intermediate write operations. At the completion of the experiment, the trace files must be collected and transferred to the ETADS host for post-processing (i.e. input to the dynamic analysis tools). The final file transfer is an important part of the process that can either incorporate a high degree of automation or can rely on the manual intervention of the human user. As a consequence of the distributed nature of the generated files, the manual option is not an attractive alternative as it increases the burden on the user to perform mundane operations and introduces the potential for error. The alternative approached designed for ETADS implementation includes the automated file retrieval for all of the trace files generated during the experiment. This alternative is entirely feasible using a combination of existing SADCA instrumentation techniques and modern network communications technology. ETADS can insert special software probes, in addition to the various other software probes already used, to pass a message from a given process back to the ETADS host process to indicate where the trace file is located. This action would be performed immediately prior to process termination and would provide the basis for trace file retrieval in an automated fashion with no direct input from the user. The ETADS host process will have sufficient information to retrieve all generated trace files.

The second approach is applicable when nodes of the distributed network do not have local file system services. During test case execution, each process is responsible for writing trace data to a network I/O stream. The instrument processing software must be designed to automatically handle the open-close network I/O operations and perform the intermediate write operations. At the completion of the experiment, no further action is required by an individual process of the SUT. During the test case execution, the ETADS host process is responsible for receiving and recording the trace data records as they are received across the network I/O streams. Thus, the global trace file is constructed dynamically as the test case is being executed.

The third approach, the hybrid approach, is the one that is most likely to be in general use in operational ETADS environments as it handles the more general case in which the heterogenous distributed network has a wide variety of available computing resources, each of which has its advantages and limitations which may not include the availability of local file system services. This approach would included characteristics of both approaches described above. The primary differences are contained in the design of the ETADS host process in which trace data is received. It must be designed to receive trace data over network I/O streams in response to requests initiated by the individual processes in the SUT. In addition, at the end of test case execution, it must operate using the first approach to collect trace files from the remaining subset of nodes forming the distributed system. The combined set of trace data is then merged and sorted and input to the ETADS dynamic analysis process

6.0 Summary of Conclusions And Recommendations

In this chapter, we will provide a summary description of the conclusions derived from the Phase I research effort and further elaborate on the rationale for a recommended course of action. For the reader's convenience, all of these detailed results are summarized in tabular form ordered according to the priority assigned to each candidate technology. Further discussion of the rationale for prioritization and details of each technique follow the summary table. Finally, we provide a summary discussion of the feasibility issues with respect to near-term extension of SADCA technology to incorporate ETADS functionality. As will be seen, the conclusions we draw from Phase I research is that it seems feasible to implement important new software testing support for the distributed real-time software domain within the following high-payoff areas:

- Dynamic Coverage Analysis
- Deterministic Execution Replay
- Timing Analysis
- Regression Testing
- Reengineering Support

In order to prioritize the selection of candidate techniques for incorporation into ETADS, OTI developed a set of criteria with which each technique was evaluated. Each criteria is rated on a scale of High-Medium-Low. In identifying and defining the meaning of the criteria used in the ratings, care was exercised in defining the terms such that there is consistent meaning associated with High-Medium-Low where High indicates a relatively "good" outcome and Low indicates a relatively "bad" outcome. Finally, considering the combined impact of all criteria, where criteria 2 and 4 have the most impact, OTI rated each technique's overall priority on a numeric scale with the value of 1 being the highest rating as shown in the table below. The criteria are:

(1) level of automatability	(4) applicability to large-scale systems
(2) applicability to distributed software	(5) applicability to real-time software
(3) effectiveness as a testing technique	(6) ease of implementation

Table 6-1 Prioritization of Techniques

	CRITERION						OVERALL PRIORITY
TECHNICAL AREA	1	2	3	4	5	6	
Minimization of Probe Effect	H	H	H	M	M+	M+	1
Deterministic Execution Testing	M	H+	M	H	M	M	1
Timing Analysis	M+	H	H	H	M	M	2
Control Flow Testing	M+	M	H	M	M	M+	3
Dependency Analysis	M	H	H	H	M+	M+	3
Network Communication Testing	H	M+	M+	H	M+	M+	4

Table 6-1 Prioritization of Techniques

	CRITERION						OVERALL PRIORITY
TECHNICAL AREA	1	2	3	4	5	6	
Data Flow Analysis	M	M	M	M	L	L	5
Problem Tracking Facility	H	M	M	M	M	H	5
Mutation Analysis	M	M	M	L	L	H	6
Static Concurrency Testing	L	L	M	L	L	L	7

Consideration of the importance of each evaluation criteria is, by definition, performed in a relative fashion. Each of the listed technical areas is considered relative to the others and assigned a relative weighting. Since the nature of the evaluation process has sufficient unknowns to preclude the complete elimination of subjective criteria, the rankings are admittedly subjective in nature. However, these evaluations are fundamentally based on our extensive experience in developing mainstream tools to support software testing.

Further consideration was also given to documenting a formal approach to combining the outcomes from each of the criteria rankings to yield the overall priority ranking. For example, the overall priority might be described as a linear combination of the criterion where the High-Medium-Low values were assigned a numerical weight. However, in view of the remaining subjectivity in selecting criteria rankings, such an approach is of dubious value and would merely serve to obscure the fact that the rankings are still subjective in nature.

Criteria 1 - level of automatability: Each technique is considered from the perspective of how much automation can be achieved in the implementation. If the algorithm or technique can be defined such that the resulting output is succinct and positively correlates with software quality, then a high score is given with respect to automatability. In contrast, a technique whose application still requires a large amount of human intervention in the process and/or human analysis and interpretation of the output would receive a low score.

Criteria 2 - applicability to distributed software: Each technique is considered as to how well the unique issues of the distributed environment are addressed. For example, a software testing technique that is quite useful in the sequential software domain could still receive a low score if the technique does not effectively address any of the special problems of distributed software testing.

Criteria 3 - effectiveness as a testing technique: Each technique is considered from the perspective of the anticipated effectiveness at detecting and reporting software defects in a manner that is conducive to human comprehension and correction of such defects. In some cases it may be necessary to consider the degree to which a given technique is likely to identify existing software defects. In other cases it may be necessary to consider the degree to which the totality of information reported by a technique obscures actual software defect reports. Higher scores are given to techniques that are useful in definitively reporting software defects.

Criteria 4 - applicability to large-scale systems: In consideration of each technique, separate evaluation scores are derived based on the degree to which the technique is scalable. The scalability criteria is a measure of how well the technique applies to large-scale systems.

Criteria 5 - applicability to real-time software: Each technique is evaluated with respect to its anticipated utility in a real-time environment. Of critical importance in evaluating this criteria is the ability of the technique to deal with the real-time performance aspects of the SUT. In this area, minimization of perturbations is very important and ability to reduce effects of non-deterministic behavior is important.

Criteria 6 - ease of implementation: Evaluation according to this criteria is designed to focus solely on the magnitude of the task of implementing the given technique. Evaluation of this criteria is purposely designed to be independent of the other considerations and is intended to serve as the principle weighting factor that accounts for effort required to implement the given technique.

6.1 Technique: Minimization of Probe Effect

It is important to note that this technology area was not originally described as one of the major areas of functionality, and yet the final result has been to elevate its status to be listed as one of the principle candidate techniques. This effect is a consequence of the relative importance of probe effects on the achievable test results gathered by ETADS due to the fact that virtually all other candidate techniques are to some degree dependent, either directly or indirectly, on how well the probe effect minimization is achieved.

Automated probe insertion is one of the principal tasks of the SADCA based technology from which ETADS is derived. Experience with SADCA has demonstrated the feasibility of the general technique. It remains for ETADS to extend the probe insertion process to provide for intelligent selection of particular probes in response to the user's selection of a probe insertion strategy. Thus, a high level of automatability is anticipated.

The applicability to distributed software is clearly high (i.e. H) in consideration of the potential effects of timing relationships on performance. In addition, it is important to consider the effects of using various types of system resources including: CPU, memory, secondary storage, and network bandwidth.

A high ranking (i.e. H) for effectiveness as a testing technique due to its general importance as an underlying technology throughout ETADS processes.

Applicability to large-scale systems is ranked (M) because it is relatively neutral in its support for such systems.

The applicability to real-time software is ranked (M+) because the technology is very important for real-time software testing, however, within this domain the software probe insertion technique has the drawback of introducing perturbation of the timeline.

The ease of implementation is ranked (M+) because, after the general strategy for probe insertion has been defined the actual modifications to the existing probe insertion process is expected to be a relatively uncomplicated activity.

The overall high ranking of this technique, (i.e. ranking -> 1) is largely due to its relatively high rankings in each individual category. However, additional consideration is also given to the dependency between this technique and the other techniques.

6.2 Technique: Deterministic Execution Testing

The deterministic execution testing service is anticipated to be a high payoff technology as it provides significant support to mitigate the effects of non-determinism.

The level of automatability is ranked (M) because a multistep process with human intervention is involved. Although the tool can be configured to analyze the software and provide the requisite instrumented versions of the SUT upon demand, multiple user operations are still required to perform the experiments and process the trace data files.

The applicability to distributed software is ranked (H+) because the technique explicitly addresses the non-deterministic behavior of concurrent distributed processes.

The effectiveness as a testing technique is ranked (M) because it provides strong support for software debugging and provides the foundation for reproducible testing.

The applicability to large-scale systems is ranked (H) because its support for reducing non-deterministic behavior during testing. Large systems are difficult to handle and test without automated control mechanisms. Deterministic Execution is applicable to a concurrent Ada program running on a single processor as well as a highly distributed set of processes executing across a WAN.

The applicability to real-time software is ranked (M) as a consequence of the expectation of higher complexity for real-time software support. Increased requirements are anticipated with respect to recording and storing all of the additional real-time system inputs necessary to support replay. In addition, time triggered values may be difficult to reproduce as part of the input.

The ease of implementation is ranked (M) because there is considerable complexity involved in developing all of the new software required to implement the technique. It requires the automation of a large number of sophisticated techniques to recognize a broad range of events and force their recurrence upon demand, and in the same sequence, in the target environment.

The overall high ranking of this technique, (i.e. ranking -> 1) correlates with the severity of the problems encountered with non-deterministic execution in distribute software, the attainable benefit from automated support, and the identification of a sound strategy for providing automated support.

6.3 Technique: Timing Analysis

The level of automatability is ranked (M+) because it compares very favorable to some of the other techniques that require additional manual steps by the user. The desired timing analysis displays can be constructed based on the time stamped event data collected in the trace files.

The applicability to distributed software is ranked (H) because this technology directly addresses an area (i.e. non-deterministic behavior) in which a large portion of the more severe software testing problems originate.

The effectiveness as a testing technique is ranked (H) because its application in the distributed software domain will greatly facilitate the identification and removal of timing-related problems.

The applicability to large-scale systems is ranked (H) because the timing analysis facilities contain sufficient flexibility in the level of detail presented in any particular report to cover the broad range of required reporting capabilities. For example, high level abstract views for large scale system view and low level detailed views to focus on behavior within a specific subsystem.

The applicability to real-time software is ranked (M) as a consequence of the trades in observability versus perturbation of the experiment. The support for experiment observability is of critical importance in testing real-time software and is of extremely high benefit. However, this benefit is tempered by the recognition of the inevitable perturbation of the SUT behavior as a consequence of applying portable software probe based technique (i.e. instead of attempting to rely on expensive, non-portable, non-scalable, and generally infeasible hardware probe techniques).

The ease of implementation is ranked (M) because the design and construction of additional software components is only of moderate complexity in comparison to the requirements of other ETADS functionality.

Timing analysis receives an overall ranking of (2), primarily because it provides major benefit in the evaluation of system behavior with respect to the major issue in the distributed software domain: non-deterministic behavior.

6.4 Technique: Control Flow Testing

The level of automatability is ranked (M+) because it is known to be comparable to the automated SADCA control flow testing capabilities and requires the same sequence of manual steps in executing test cases and post-processing the trace files.

The applicability to distributed software is ranked (M) because it can be readily extended to distributed software testing through minor extensions to the set of common language primitives.

The effectiveness as a testing technique is ranked (H) because it is comparable to the techniques currently implemented in SADCA and METAsoft which have demonstrated the feasibility and utility of the technique through application on a variety of software development projects including mission critical software embedded in missiles and ground launch control software.

The applicability to large-scale systems is ranked (M) because the technology, as defined in ETADS, is readily extensible to large-scale distributed systems. This relatively high ranking is assigned with the knowledge that the interprocess control flow analysis in the form of concurrency state coverage metrics is not included in the planned functionality.

The applicability to real-time software is ranked (M) because it can be readily applied to real-time software with the caveat that the user must understand and accept the limitations of the software probe based technology. Minor perturbations of the SUT are to be expected to some extent and can be minimized through the careful selection of instrumentation strategy through the ETADS user interface.

The ease of implementation is ranked (M+) because it builds on existing SADCA technology with extensions to the set of common language primitives and the associated graphical display mechanisms.

Control Flow Testing receives an overall ranking of (3) because of the combination of ease of implementation, demonstrated feasibility in the sequential software domain, and extensibility to the distributed systems domain.

6.5 Technique: Dependency Analysis

The level of automatability is ranked no higher than (M) because the ability to identify process relationships in a static analysis process is limited for loosely integrated communicating processes as is the general case in the distributed processing environment. Although dynamic analysis data can provide a large amount of additional data for dynamic dependency analysis, the static analysis of dependency relationships is frequently limited by the inability to follow the use of process identification variables, arrays, and/or pointers in the originating source without performing symbolic execution.

The applicability to distributed software is ranked (H) because the focus of the technique is to provide design information that is critical to the understanding and evaluation of correct system behavior.

The effectiveness as a testing technique is ranked (H) because the availability of the dependency data and its graphical rendition for human interpretation can greatly aid the user in the identification of error conditions.

The applicability to large-scale systems is ranked (H) because the focus of the technique is to provide design information that is critical to the understanding and evaluation of correct behavior and is readily scalable to large-scale systems.

The applicability to real-time software is ranked (M+) because, like large-scale systems, it places the focus of analysis on an aspect of the behavior that is critical to evaluation of correct behavior.

The ease of implementation is ranked (M+) overall because the dynamic aspect of dependency analysis is relatively high, but the static aspect of dependency analysis required to achieve more than a minimal amount of information is relatively low. Thus, the M+ ranking is a composite of the two components of dependency analysis.

Dependency analysis receives an overall ranking of (3) because it addresses an aspect of the distributed software design and behavior evaluation that can provide major benefit in the identification and subsequent elimination of software defects.

6.6 Technique: Network Communication Testing

The level of automatability is ranked (H) because the additional software support required to implement the desired functionality is of relatively low complexity and builds on proven software probe insertion and post processing techniques.

The applicability to distributed software is ranked (M+) because the techniques focus on a critical aspect of distributed software behavior. However, its ranking is compared to the even higher expected benefits to be derived from techniques such as timing analysis and deterministic execution replay.

The effectiveness as a testing technique is ranked (M+) because it is a very useful technique for performance evaluation. However, its limited scope also means that it can be effective only within its predefined analysis scope and will therefore be of somewhat less overall effectiveness than a more general technique such as deterministic execution replay.

The applicability to large-scale systems is ranked (H) because the technique can be applied to increasingly larger systems with a high degree of effectiveness. Furthermore, in view of the general lack of such capability in the commercial world, this technology area deserves a very high priority ranking.

The applicability to real-time software is ranked (M+). The motivation for this ranking is comparable to that for applicability to large-scale systems. However, real-time perturbation effects slightly diminish the expected benefit.

The ease of implementation is ranked (M+) because the technique can be implemented primarily through the reorganization and processing of existing software probe based trace data (i.e. assuming concurrent implementation with other ETADS techniques such that the time stamped trace data exists).

Network communications testing is ranked (4) not because of low expected benefit, but because the expected benefits of other techniques are so high. Evaluation of this technique on its own merits indicates an overall high priority ranking as a consequence of its versatility across a range of system sizes and its ability to assist the identification of errors.

6.7 Technique: Data Flow Analysis

The level of automatability is ranked (M) because this is a relatively complicated algorithmic process to be designed and implemented. Previous experience with SADCA data flow analysis technology in a multilingual environment firmly established certain complexities in the identification of variable usage. Extending the utility of data flow analysis to track the utilization of process identification variables will be an important supporting technology for dependency analysis. However, extending data flow analysis between processes will be extremely difficult in general (due to the message passing communications paradigm prevalent in distributed systems) and will be of limited utility in identifying existing errors. Extending the utility of data flow analysis within the bounds of a single process is somewhat complex, but can derive significant benefits in terms of error detection and reporting capabilities.

The applicability to distributed software is ranked (M) because the potential benefit of the technique is high and data flow analysis supports dependency analysis. However, limitations on the ability to perform interprocess analysis restricts its general applicability.

The effectiveness as a testing technique is ranked (M) for largely the same reasons as for applicability to distributed software.

The applicability to large-scale systems is only ranked (M) because of the limitations of interprocess data flow analysis. However, the limitation can be considered in both a restrictive and beneficial sense. It is restrictive in the sense that interprocess data flow analysis across a large-scale network of processes is very limited in the ability to detect errors. In contrast, the ability to perform intra-process data flow analysis scales in a linear fashion as there is (by definition) little (if any) coupling between data flow analysis processes. For example, the data flow analysis for a given process is performed independent from any other process and largely resembles the data flow analysis performed on a program basis in the sequential software testing world.

The applicability to real-time software is ranked (L) because of the limitations of data flow analysis in response to external inputs and the limitations on interprocess data flow analysis.

The ease of implementation is ranked (L) primarily because of the complexity of the algorithms required by this service. Intra-process data flow analysis is complex. interprocess data flow analysis is, where technically feasible, extremely complex.

Data flow analysis receives an overall ranking of (5) primarily as a result of its support for dependency analysis and intra-process data flow analysis. In addition, the ability to detect and report errors can be expected to provide greater benefit that provide by other techniques such as Problem Tracking and Reporting.

6.8 Technique: Problem Tracking Facility

The level of automatability is ranked (H) because the complexity of designing and implementing the additional DBMS services and computational services is relatively low and can be completely automated.

The applicability to distributed software is ranked (M) because it can be designed to operate on distributed software, although the technique is not limited to this domain.

The effectiveness as a testing technique is ranked (M) because it has good potential to guide the long term development and improvement of the software testing process.

The applicability to large-scale systems is ranked (M) because it is a general technique, whose applicability is not limited by the size of the SUT. However, the technique does not provide any unique technical support that specifically targets large-scale systems. On the other hand, the magnitude of the software testing task for large-scale systems can be taken as a strong argument for any tool that helps to focus software testing effort on high payoff techniques.

The applicability to real-time software is ranked (M) because its is general technique that may be beneficial in this domain but does not provide any unique technical support the specifically targets real-time software.

The ease of implementation is ranked (H) because the complexity of the required software is anticipated to be very low.

Problem Tracking Facility receives a lower overall ranking of (5) because its anticipated benefits can be expected to be derived in long-term as opposed to the immediate benefits expected from the higher ranked techniques such as Deterministic Execution Replay.

6.9 Technique: Mutation Analysis

The level of automatability is ranked (M) because there is a certain degree of user interaction required in the current level of ETADS mutation analysis support. ETADS can provide automated support for the insertion of user specified mutants and the subsequent construction of mutant versions of the SUT.

The applicability to distributed software is ranked (M) because its focus does not directly address the primary problems in testing distributed software.

The effectiveness as a testing technique is ranked (M) because its focus does not directly address the primary problems in testing distributed software.

The applicability to large-scale systems is ranked (L) because the mutation analysis process requires the execution of a relatively large number of test cases with different mutants in order to be most effective and this is more difficult to accomplish for large-scale systems.

The applicability to real-time software is ranked (L) because the effects of real-time non-deterministic behavior complicate the application of mutation testing. Effective application of this technique requires the combination of deterministic execution testing, which while feasible and supported by ETADS is more resource intensive throughout the software testing process to yield the desired benefit.

The ease of implementation is ranked (H) because the complexity is low for implementing the technique within the ETADS tool.

Mutation Analysis receives an overall ranking of (7) because it has less immediate tangible benefit than the other techniques and is a more difficult testing strategy to implement for large distributed real-time systems.

6.10 Technique: Static Concurrency Testing

The level of automatability is ranked (L) because current techniques for evaluating the concurrency state space are incapable of handling the explosion in state space size anticipated for real-world systems. The technique is useful in theory, but suffers greatly from real-world limitations.

The applicability to distributed software is ranked (L) due to the explosive growth in size of the concurrency state space using the known methods of analysis. Additional research is necessary to identify techniques for managing the problem size.

The effectiveness as a testing technique is ranked (M) as a consequence of its potential for identifying and reporting problems with the utilization of concurrency constructs that may lead to anomalies such as deadlock.

The applicability to large-scale systems is ranked (L) primarily because of the complexity of implementation. Although the technique can, in principle, detect potential anomalies in the concurrency states space (e.g. deadlock states), the technical complexity of implementation overshadow the potential benefits.

The applicability to real-time software is ranked (L) for the same reasons as for applicability to large-scale systems.

The ease of implementation is ranked (L) primarily because of the explosive growth in problem size as the size of the SUT increases. In contrast, the complexity of some algorithms such as Taylor's algorithm for synchronization sequence analysis is relatively low and would not be very difficult to implement. However, actual execution for moderate to large problems would experience excessive memory and CPU resource consumption at run-time.

Static Concurrency Testing receives an overall ranking of (7) as a result of the expected exponential growth of the algorithms.

A. Appendix A - Bibliography

Abrams, Mark, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains," IEEE Transactions on Parallel and Distributed Systems, vol. 3, p. 672, November 1992.

Abstreiter, F., "Visualizing and Analysing the Runtime Behavior of Parallel Programs," CONPAR '90: Joint International Conference on Vector and Parallel Processing, pp. 828-839, 1990.

Agrawal, H., DeMillo, R.A., and Spafford, E., "An ExecutionBacktracking Approach to Debugging," IEEE Software, p. 21, 1991.

Andersland, M.S. and Casavant, T.L., "Recovering Uncorrupted Event Traces from Corrupted Event Traces in Parallel/Distributed Computing Systems," 1991 International Conference on Parallel Processing, vol. II, p. 108, 1991.

Annaratone, M., "MPPs, Amdhal's Law, and Comparing Computers," IEEE, p. 465, 1992.

Araki, K., Furukawa, Z., and Cheng, J., "A General Framework for Debugging," IEEE, p. 14, May 1991.

Carver, R. and Tai, K.C., "Deterministic Execution Testing of Concurrent Ada Programs," ACM, p. 528, 1989.

Carver, R.H. and Tai, K.C., "Replay and Testing for Concurrent Programs," IEEE Software, pp. 66-74, March 1991.

Carver, R.H., "Mutation-Based Testing of Concurrent Programs," Proceedings of the International Test Conference, 1994.

Chen, Y., Tsai, W.T., and Chao, D., "Dependency Analysis - A Petri Net Based Technique for Synthesizing Large Concurrent Systems," IEEE Transactions on Parallel and Distributed Systems, vol. 4, p. 414, April 1993.

Choi, B.J., Mathur, A.P., High Performance Mutation Testing, The Journal of Systems and Software, 20(2), pp135-152, February 1993

DeMillo, R. A., *Program Mutation: An Approach to Software Testing*, Georgia Institute of Technology report prepared for Army Research Office, RTP, NC, Contract Nos.: ONR-N00014-79-C-0231 and ARO-DAAG29-80-C-0120, April 1983.

DeMillo, R.A. and Offutt, Constraint -Based Automatic Test Data Generation, IEEE Transactions on Software Engin-erring, 17(9), pp 900-910, September 1991

DeMillo, R.A. et al, Compiler-Integrated Program Mutation, Proceedings of the Fifteenth Annual International Computer Software and Applications Conference (COMPSAC), pp 351-356, Tokyo, Japan, September 11-13 1991, IEEE Computer Society Press

Dinning, A., "A Survey of Synchronization Methods for Parallel Computers," IEEE Computer, pp. 66-77, July 1989.

Emrath, P.A., Ghosh, S., and Padua, D.A., "Detecting Nondeterminacy in Parallel Programs," IEEE Software, pp. 69-77, January 1992.

Gehani, N.H. and Roome, W.D., "Rendezvous Facilities: Concurrent C and the Ada Language," IEEE Transactions on Software Engineering, vol. 14, p. 1546, November 1988.

Hariri, S., Ladan, M., and Kee, K.K., An Efficient Approach for Detecting Data Races in Parallel Programs.

Helmbold, D.P., McDowell, C.E., and Wang, J., "Analyzing Traces with Anonymous Synchronization," 1990 International Conference on Parallel Processing, vol. II, pp. 70-77, 1990.

Hollingsworth, J.K., Irvin, R.B., and Miller, B.P., "The Integration of Application and System Based Metrics in a Parallel Program Performance Tool," Proc. of the 1991 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1991.

Hollingsworth, J.K., Miller, B.P., and Cargille, J., "Dynamic Program Instrumentation for Scalable Performance Tools," Proceedings of the 1994 Scalable High Performance Computing Conference, May 1994.

Howden, W.E., "The Theory and Practice of Functional Testing," IEEE Software, 1985.

Irvin, R.B. and Miller, B.P., "Multi-Application Support in a Parallel Program Performance Tool," Technical Report, Univ. of WisconsinMadison, 1993.

Knight, J.C. and Urquhart, J.I.A., "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems," IEEE Transactions on Software Engineering, vol. 13, p. 553, May 1987.

Krauser, E.W., Mathur, A.P., and Rego, V.J., "High Performance Software Testing on SIMD Machines," IEEE Transactions on Software Engineering, vol. 17, p. 403, May 1991.

Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, vol. 21, p. 558, July 1978.

Leblanc, T.J. and Mellor-Crummey, J.M., "Debugging Parallel Programs with Instant Replay," IEEE Transactions on Computers, vol. C-36, pp. 471-482, April 1987.

Lichtman, Z.L., "Generation and Consistency Checking of Design and Program Structures," IEEE Transactions on Software Engineering, vol. 12, p. 172, January 1986.

Long, D.L. and Clarke, L.A., "Task Interaction Graphs: An Intermediate Representation for Concurrency Analysis," Technical Report 88-47, Computer and Information Science, University of Massachussetts at Amherst, 1988.

Long, D.L. and Clarke, L.A., "Task Interaction Graphs for Concurrency Analysis," ACM, p. 44, 1989.

Long, D.L., Clarke, L.A., and Fialli, J., "Ada Language Considerations for Concurrency Analysis," Technical Report 8942, Computer and Information Science, University of Massachussetts at Amherst, 1989.

Long, D. and Clarke, L.A., "Data Flow Analysis of Concurrent Systems that use the Rendezvous Model of Synchronization," ACM, pp. 236-250, 1991.

Lopez-Benitez, N., "Dependability Modeling and Analysis of Distributed Programs," IEEE Transactions on Software Engineering, vol. 20, p. 345, May 1994.

Lumpp, J.E., Gannon, J.A., Andersland, M.S., and Casavant, T.L., "A Technique for Recovering from Software Instrumentation Intrusion in Message-Passing Systems," Technical Report, Parallel Processing Laboratory, Electrical and Computer Engineering Department, University of Iowa, April 1992.

Melamed, B. and Morris, R.J.T., "Visual Simulation: The Performance Analysis Workstation," IEEE, p. 87, August 1985.

Miller, B. P., "A Measurement System for Distributed Programs," *IEEE Transactions on Computers*, Vol. 37, No. 2, February 1988, pp. 243-248.

Miller, B.P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S., and Torzewski, T., "IPS-2: The Second Generation of a Parallel Program Measurement System," IEEE Transactions on Parallel and Distributed Systems, vol. 1, pp. 206-217, April 1990.

Morell, L.J., "A Theory of Fault-Based Testing," IEEE Transactions on Software Engineering, vol. 16, p. 844, August 1990.

Netzer, R.H.B. and Ghosh, S., "Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization," 1992 International Conference on Parallel Processing, vol. II, p. 242, 1992.

Netzer, R.H.B and Miller, B.P., "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," Supercomputing '92, 1992.

Nichols, K.M. and Edmark, J.T., "Modeling Multicomputer Systems with PARET," IEEE, p. 39, May 1988.

Notomi, M. and Murata, T., "Hierarchically Organized Petri Net State Space for Reachability and Deadlock Analysis," IEEE, p. 616, 1992.

Notomi, M. and Murata, T., "Hierarchical Reachability Graph of Bounded Petri Nets for Concurrent-Software Analysis," IEEE Transactions on Software Engineering, vol. 20, p. 325, May 1994.

Offutt, A.J. and Lee, S.D., "An Empirical Evaluation of Weak Mutation," IEEE Transactions on Software Engineering, vol. 20, p. 337, May 1994.

Olsson, R.A., Crawford, R.H., Ho, W.W., and Wee, C. E., "Sequential Debugging at a High Level of Abstraction," IEEE Software, p. 27, 1991.

Omar, A.A. and Mohammed, F.A., "A Survey of Software Functional Testing Methods," ACM SIGSOFT Software Engineering Notes, vol. 16, p. 75, April 1991.

Pancake, C.M. and Netzer, R.H.B., "A Bibliography of Parallel Debuggers, 1993 Update," Technical Report, 1993.

Paoli, De F. and Morasca, S., "Extending Software Complexity Metrics to Concurrent Programs," Proceedings of the 14th An nual International Computer Software and Applications Conference, pp. 414-419, 1990.

Papelis, Y.E. and Casavant, T.L., "XPAT: An Interactive Graphical Tool for Synthesis of Concurrent Software using Petri Nets," 1991 International Conference on Parallel Processing, vol. II, p. 292, 1991.

Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views," IEEE Transactions on Software Engineering, vol. 11, p. 276, March 1985.

Rubin, R.V., Walker, J., and Golin, E.J., "Early Experience with the Visual Programmer's Workbench," IEEE Transactions on Software Engineering, vol. 16, p. 1107, October 1990.

Schatz, E. and Ryder, B.G., "Directed Tracing to Detect Race Conditions," 1992 International Conference on Parallel Processing, vol. II, p. 247, 1992.

Schutz, W., "On the Testability of Distributed Real-Time Systems," 1991 IEEE 10th Symposium on Reliable Distributed Systems, pp. 52-61, 1991.

Schwan, K. and Matthews, J., "Graphical Views of Parallel Programs," ACM SIGSOFT Software Engineering Notes, vol. 11, p. 51, July 1986.

Sha, L. and Goodenough, J.B., "Real-Time Scheduling Theory and Ada," IEEE, p. 53, April 1990.

Shatz, S.M., Mai, K., Black, C., and Tu, S., "Design and Implementation of a Petri Net Based Toolkit for Ada Tasking Analysis," IEEE Transactions on Parallel and Distributed Systems, vol. 1, pp. 424-441, October 1990.

Shimomura, T. and Isoda, S., "Linked-List Visualization for Debugging," IEEE, p. 44, May 1991.

Snyder, L., "Parallel Programming and the Poker Programming Environment," IEEE, p. 27, July 1984.

Stasko, J.T., "Tango: A Framework and System for Algorithm Animation," IEEE, September 1990.

Tai, K.C., "On Testing Concurrent Programs," *Proceedings of COMPSAC '85: Computer Software and Applications Conference*, 1985, pp. 310-317.

Tai, K.C. and Obaid, E.E., "Reproducible Testing of Ada Tasking Programs," IEEE Second International Conference on Ada Applications and Environments, pp. 69-79, 1986.

Tai, K.C. and Ahuja, S., "Reproducible Testing of Communication Software," *Proceedings of COMPSAC '87: Computer Software and Applications Conference*, 1987, pp. 331-337.

Tai, K.C., Carver, R.H., and Obaid, E.E., "Debugging Concurrent Ada Programs by Deterministic Execution," IEEE Transactions on Software Engineering, vol. 17, p. 45, January 1991.

Tai, K.C. and Carver, R.H., "Testing of Distributed Programs," Chapter in Handbook of Parallel and Distributed Computing, June 3, 1994.

Tsai, J. J. P., Fang, K. Y., Chen, H. Y., and Bi, Y. D., "A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, August 1990, pp. 897-916.

Taylor, R.N. and Osterweil, L.J., "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Transactions in Software Engineering, vol. SE-6, pp. 265-278, May 1980.

Tay[illegible] N., "A General-Purpose Algorithm for Analyzing Concurrent Programs," ACM, 1983.

Taylor, [illegible]., Levine, D.L., and Kelly, C.D., "Structural Testing of Concurrent Programs," IEEE Transactions on [illegible]oftware Engineering, vol. 18, pp. 206-215, March 1992.

Tso, K.S., Hecht, M., and Littlejohn, K., "Complexity Metrics for Avionics Software," Proceedings of the IEEE 1992 National Aerospace and Electronics Conference, pp. 603-609, 1992.

Young, M., Taylor, R.N., Levine, D.L., Forester, K., and Brodbeck, D., "A Concurrency Analysis Tool Suite: Rationale, Design, and Preliminary Experience," SERC Technical Report TR-128-P, Software Engineering Research Center, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, October 1992.

B. Appendix B - Terms and Abbreviations

This appendix of the final report is intended as an aid in the understanding of this document. It contains a definition of specific terms and abbreviations used within the context of this report.

- Concurrent Software - a software configuration in which multiple processes are executed in the target environment at the same time, either on multiple interconnected machines, or on the same machine under some form of time-sharing scheme
- Chart - a control flow diagram (i.e. flow chart) corresponding to a specific subroutine from the originating source code; a general term referring to the composite entity, and its attributes, from which a flowchart diagram is drawn
- DBMS - DataBase Management System
- Dynamic Analysis - The general category of code analysis techniques designed to identify the run-time characteristics of the software under test.
- ETADS - Environment for Test and Analysis of Distributed Software
- Instrument - (noun) synonymous with Software Probe; a specially formatted piece of code inserted into the SUT to record a specific piece(s) of run-time information
- Instrument - (verb) the process of inserting instruments/software probes into the SUT
- LAN - local area network
- MPI - Message Passing Interface
- PVM - Parallel Virtual Machine
- SADCA - Static and Dynamic Code Analyzer
- Static Analysis - The general category of code analysis techniques designed to identify structural characteristics of the software under test.
- Software Probe - a specially formatted piece of code inserted into the SUT to record a specific piece(s) of run-time information
- SUT - Software Under Test
- TIE - Task Interaction Event
- Trace - a sequence of event data (may or may not have time stamps)
- Trace File - a file containing a collection of "Trace" data
- WAN - wide area network